Qualified Types with Boolean Algebras

EDWARD LEE, University of Waterloo, Canada
JONATHAN LINDEGAARD STARUP, Aarhus University, Denmark
ONDŘEJ LHOTÁK, University of Waterloo, Canada
MAGNUS MADSEN, Aarhus University, Denmark

We propose type qualifiers based on Boolean algebras. Traditional type systems with type qualifiers have been based on lattices, but lattices lack the ability to express *exclusion*. We argue that Boolean algebras, which permit exclusion, are a practical and useful choice of domain for qualifiers.

In this paper, we present a calculus System $\mathsf{F}_{<:\mathsf{B}}$ that extends System $\mathsf{F}_{<:}$ with type qualifiers over Boolean algebras and has support for negation, qualifier polymorphism, and subqualification. We illustrate how System $\mathsf{F}_{<:\mathsf{B}}$ can be used as a design recipe for a type and effect system, System $\mathsf{F}_{<:\mathsf{BE}}$, with effect polymorphism, subeffecting, and polymorphic effect exclusion. We use System $\mathsf{F}_{<:\mathsf{BE}}$ to establish formal foundations of the type and effect system of the Flix programming language. We also pinpoint and implement a practical form of subeffecting: abstraction-site subeffecting. Experimental results show that abstraction-site subeffecting allows us to eliminate all effect upcasts present in the current Flix Standard Library.

CCS Concepts: • **Software and its engineering** → **Data types and structures**; **Semantics**; **Compilers**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Type Systems, Type Qualifiers, Boolean Algebras, System $\mathsf{F}_{<:}$, Flix

ACM Reference Format:
Edward Lee, Jonathan Lindegaard Starup, Ondřej Lhoták, and Magnus Madsen. 2025. Qualified Types with Boolean Algebras. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 318 (October 2025), 27 pages. https://doi.org/10.1145/3763096

1 Introduction

Type qualifiers enrich type systems by decorating each type with additional information about values, behavior, or computation [15]. For example, type qualifiers have been used to express aliasing, immutability, capture tracking, nullability, reachability, among others [7, 39, 41, 51]. Type qualifiers have also been used in type and effect systems [38], of which there has been much recent work [9, 30, 32, 35, 40, 48].

An important design choice is the language of qualifiers and whether it supports subqualification (i.e., subtyping but for qualifiers) and qualifier polymorphism (i.e., polymorphism over qualifiers). Traditionally, most qualifier systems have been based on lattices which can support both [25]. However, lattices are not always expressive enough and lack a story for type inference in the general case. In this paper, we propose *type qualifiers over Boolean algebras* supporting exclusion (i.e. *negation*) while retaining qualifier polymorphism and subqualification. We find that exclusion meaningfully increases expressive power compared to qualifier systems based on lattices.

Authors' Contact Information: Edward Lee, University of Waterloo, Waterloo, ON, Canada, e45lee@uwaterloo.ca; Jonathan Lindegaard Starup, Aarhus University, Aarhus, Denmark, jls@cs.au.dk; Ondřej Lhoták, University of Waterloo, Waterloo, ON, Canada, olhotak@uwaterloo.ca; Magnus Madsen, Aarhus University, Aarhus, Denmark, magnusm@cs.au.dk.

ACM 2475-1421/2025/10-ART318
https://doi.org/10.1145/3763096

For example, in a type and effect system, we may want to express that a higher-order function handles an exception. In the proposed system, which is implemented in Flix, we can express this as:

```
def handleException[a, ef](f: Unit -> a \ ef): a \ (ef - {DivByZero})
```

Here `handleException` is polymorphic over the type variable `a` and the effect variable `ef`. The function parameter is of type `Unit -> a` and has effect `ef`, which we treat as a qualifier on its type. Because `handleException` is effect polymorphic, we can instantiate the effect variable `ef` to any effect. For example, it could be the effect set `{Print, DivByZero}`. The effect of the call to `handleException` is then `{Print, DivByZero} - {DivByZero} = {Print}`. The reason this works is because the language of effects forms a Boolean algebra and subtraction can be expressed as intersection with complement, i.e. $P - Q = P \cap Q^{\complement}$. But we still want (and can show!) that `ef - {DivByZero}` $\subseteq$ `ef`.

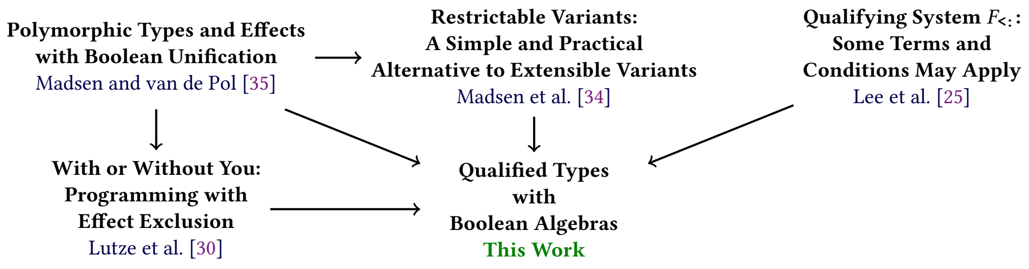
In summary, the contributions of this paper are:

- **(Calculus)** We present System $\mathsf{F}_{<:\mathsf{B}}$, a calculus that extends System $\mathsf{F}_{<:}$ with type qualifiers over Boolean algebras with support for qualifier polymorphism and subqualification. We show how System $\mathsf{F}_{<:\mathsf{B}}$ can be used as a design recipe for a type and effect system, System $\mathsf{F}_{<:\mathsf{BE}}$, with effect polymorphism, subeffecting, and polymorphic effect exclusion.
- **(Soundness)** We formalize System $\mathsf{F}_{<:\mathsf{B}}$ and System $\mathsf{F}_{<:\mathsf{BE}}$ using the Rocq theorem prover. We establish soundness of both systems. We leverage soundness of System $\mathsf{F}_{<:\mathsf{BE}}$ to establish the soundness of the type and effect system in the Flix programming language.
- **(Implementation)** We pinpoint a limited form of subeffecting that we call *abstraction-site subeffecting*, which is amenable to type and effect inference. We implement abstraction-site subeffecting as an extension of Flix, building on System $\mathsf{F}_{<:\mathsf{BE}}$.
- **(Evaluation)** We empirically evaluate the usefulness of abstraction-site subeffecting on the Flix Standard Library. The results show that we are able to eliminate all 47 effect upcasts.

The soundness proofs are formalized in Rocq; details are discussed in Section 3. The extended Flix compiler and the soundness proofs are available in the paper artifact [23, 24].

Research Landscape of Type Qualifiers and Boolean Algebras

We briefly discuss how this paper significantly extends recent work in type qualifiers and Boolean algebras as used in programming languages:


Polymorphic Types and Effects with Boolean Unification
Madsen and van de Pol [35]
Restrictable Variants: A Simple and Practical Alternative to Extensible Variants
Madsen et al. [34]
Qualifying System $F_{<:}$: Some Terms and Conditions May Apply
Lee et al. [25]
With or Without You: Programming with Effect Exclusion
Lutze et al. [30]
Qualified Types with Boolean Algebras
This Work

Specifically, this paper is the culmination of several lines of research:

- Madsen and van de Pol [35] introduce a polymorphic type and effect system based on Boolean formulas, use Boolean unification for inference, and implement it in Flix. The proposed system **lacks subeffecting** and **lacks a mechanized proof.**
- Lutze et al. [30] recast the type and effect system from *Boolean formulas* to *Boolean set formulas* and introduce the idea of polymorphic effect exclusion. The proposed system also **lacks subeffecting** and **lacks a mechanized proof.**

- Madsen et al. [34] introduce a type system for restrictable variants: a form of refinement types for algebraic data types where each ADT is indexed by type-level set formulas. The proposed system **lacks a subqualification relation**, though it uses polymorphism for similar purposes. The system also **lacks a mechanized proof.**
- Lee et al. [25] introduce a framework for type qualifiers on lattice formulas. The framework extends System F, supports subqualification, and its correctness is established in Rocq. The system is **limited to lattices** (i.e., lacks support for negation of qualifiers), **lacks a story for type inference**, and **lacks an implementation.**

In this paper, we present a framework for **type qualifiers based on *Boolean algebras* with negation**. The system supports **subqualification** and is **proven correct in Rocq**. Using this system, we add a practical form of **subeffecting** to Flix, which we call **abstraction-site subeffecting**. We **implement** abstraction-site subeffecting in Flix and **empirically evaluate its usefulness**.

There have been several other approaches to integrate Boolean algebras in type systems with different tradeoffs and we discuss these in Section 6.

2 Motivation

We begin with two examples to motivate the need for type qualifiers with subqualification over Boolean algebras: a polymorphic type and effect system [30, 35] and restrictable variants [34]. Both examples benefit from polymorphism and subqualification.

2.1 A Polymorphic Type and Effect System with Effect Exclusion

Recently, Flix added *effect exclusion* to its type and effect system [30]. The idea is that a higher-order function specifies that a function argument can have *any* effect, *except* for a specific set of forbidden effects. For example, effect exclusion can be used to: (i) ensure that resources are not being inadvertently released, (ii) avoid deadlocks resulting from locks being acquired cyclically, (iii) prevent locks from being inadvertently unlocked or re-locked, (iv) prevent error handlers from throwing exceptions themselves, (v) stop iterators from modifying their underlying collection, (vi) avoid infinite recursion or looping behavior, and (vii) avoid re-entrance issues where a function must not enter itself.

To demonstrate how polymorphism, exclusion, and subeffecting emerge and interact, we present the implementation of a simple task scheduler in Flix. This scheduler enables functions to be enqueued for later execution. We begin by defining a polymorphic data structure to store the scheduled functions:

```
struct WorkList[ef: Eff, r: Region] {
    mut l: List[Unit -> Unit \ ef]
}
```

The `WorkList` struct has a single mutable field, `l`, which holds an immutable list of functions of type `Unit -> Unit \ ef`. The struct is parameterized by two type variables: `ef` and `r`. The `ef` parameter serves as an upper bound on the effects that functions in the queue may perform, while `r` designates the memory region with which the struct is associated. For a `WorkList` value `w`, the field `l` can be read using `w->l` and updated with `w->l = v`. Reading or writing the field causes a heap effect in `r`.

We can now define a function `schedule` to enqueue a new task `f`:

```
def schedule(f: Unit -> Unit \ ef - Block, w: WorkList[ef - Block, r]): Unit \ r
    = w->l = Cons(f, (w->l))
```

Here we allow `f` to have any effect *except* the `Block` effect. The idea is that we want to exclude (potentially) blocking operations from being enqueued, since such operations could stall the entire scheduler. The signature specifies that the effect of `f` must be `ef - Block`, which means that no matter how `ef` is instantiated, we cannot pass a function that has the `Block` effect. This example illustrates how *effect exclusion* can be used to both express and enforce programmer intent.

We can now schedule and execute tasks. For example, we can write:

```
def main(): Unit \ IO =
    region rc {
        let w = WorkList.empty(rc);

        // We schedule two functions of type: Unit -> Unit \ IO
        WorkList.schedule(() -> println("Hello"), w);
        WorkList.schedule(() -> println("World!"), w);

        // And we can run all scheduled functions.
        WorkList.runAll(w)
    }
```

If we try to schedule a function that has the `Block` effect:

```
def main(): Unit \ IO =
    region rc {
        let w = WorkList.empty(rc);
        // Here Console.readLine() has type: Unit -> String \ {Block, IO}
        WorkList.schedule(() -> {Console.readLine(); ()});
    }
```

We get a type and effect error:

```
>> Unable to unify the effect formulas: 'Block' and '(~Block) & ef'.

5 |         WorkList.schedule(() -> Console.readLine(), w);
                              ^^^^^^^^^^^^^^^^^^^^^^^^
                              mismatched effect formulas.
```

What happens if we try to enqueue two functions that have different effects?

For example, what if we have the functions:

```
def getCurrentUser(): String \ Env
def getCurrentHour(): Int32 \ Time
```

If we write the following in the *current* version of Flix:

```
let w = WorkList.empty(rc);
WorkList.schedule(() -> println(getCurrentUser()), w);
WorkList.schedule(() -> println(getCurrentHour()), w);
```

Surprisingly, we get a type error:

```
>> Unable to unify the effect formulas: '{Env, IO}' and '{Time, IO}'.

6 |         WorkList.schedule(() -> println(getCurrentHour()), w);
                              ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                              mismatched effect formulas.
```

The problem is a *lack of subeffecting*. During inference, at the first call to `schedule`, we infer that the type of `WorkList` is `WorkList[{Env, IO}, r]`. But this type precludes scheduling a function with the `{Time, IO}` effects, just one of the `Time` or `IO` effects, or even scheduling a pure function.

What we want is for `WorkList` to be assigned a type like `WorkList[{Env, Time, IO}, r]` and then use subeffecting to pass in functions with the effects `{Env, IO}` and `{Time, IO}`, respectively.

The Flix Standard Library contains many functions where the lack of subeffecting hurts and has required its programmers to insert explicit effect upcasts (`checked_ecast`). One example is in the `List.foldRight` function:

```
def foldRight(f: (a, b) -> b \ ef, s: b, l: List[a]): b \ ef =
    def loop(ll, k) = match ll {
        case Nil     => k(s)
        case x :: xs => loop(xs, ks -> k(f(x, ks)))
    };
    loop(l, x -> checked_ecast(x))
```

Here, `foldRight` is implemented using continuation-passing style to avoid overflowing the stack. Notably, the (inferred) effect of the continuation `k` is `ef`, from the function parameter `f`. This means that we cannot call `loop` with the identity function to start the computation, due to the lack of subeffecting.

In Section 5, we present a detailed study of the Flix Standard Library and how the lack of subeffecting in Flix has led to numerous casts throughout the library. We then refactor the Flix Standard Library to use abstraction-site subeffecting, which eliminates all casts.

2.2 Restrictable Variants

We give one more example to illustrate the usefulness of qualifiers based on Boolean algebras and of qualification: *restrictable variants* [34]. A restrictable variant is an algebraic data type, i.e., a variant type, indexed by a type-level set formula. The index over-approximates the variants in the algebraic data type that can occur. In other words, restrictable variants are refinement types [16]. Following the examples of Madsen et al. [34], we can define a restrictable variant for a data type that models Boolean expressions:

```
enum Expr[s] {
    case Var(Int32)
    case Cst(Bool)
    case Not(Expr[s])
    case Or(Expr[s], Expr[s])
    case And(Expr[s], Expr[s])
    case Xor(Expr[s], Expr[s])
}
```

The index, or qualifier, `s`, is a type-level Boolean *set formula* that ranges over the variants of the algebraic data type. The intuition is that given a type like `Expr[{Var, Cst}]`, values of that type can only use the `Var` and `Cst` constructors of `Expr` (also recursively if relevant).

```
def eval(e: Expr[~Var]): Bool =
    choose e {
        // Var case omitted: We can only evaluate closed terms.
        case Cst(b)    => b
        case Not(x)    => not eval(x)
        case Or(x, y)  => eval(x) or eval(y)
        case And(x, y) => eval(x) and eval(y)
        case Xor(x, y) => eval(x) != eval(y)
    }
```

We can use the `choose` and `choose*` pattern match constructs to write functions that only operate on a subset of the values of the restrictable variant. `choose` is used to convert a restrictable variant into some other type and `choose*` is used to transform the variant itself, keeping careful track of its index. For example, take `eval` from above, which reduce a *closed* term into a Boolean constant (`~Var` is the complement of `Var`). We have omitted the `Var` variant in the pattern match, which has been captured in the type of the argument to `eval`. Hence we cannot call `eval` with a Boolean expression that may contain a variable.

We can also write a structure-preserving map function using `choose*`:

```
def map(f: Int32 -> Int32, e: Expr[s]): Expr[s] =
    choose* e {
        case Var(x)     => Var(f(x))
        case Cst(b)     => Cst(b)
        case Not(x)     => Not(map(f, x))
        case Or(x, y)   => Or(map(f, x), map(f, y))
        case And(x, y)  => And(map(f, x), map(f, y))
        case Xor(x, y)  => Xor(map(f, x), map(f, y))
    }
```

Here we map a function over the variable names of the Boolean expression. Notably, the `map` function uses polymorphism to *relate the input type to the output type*. The signature of `map` tells us that if the input does not contain, e.g., the `Var` constructor, *then neither does the output*.

Finally, we can also write a substitution function that replaces each variable in a Boolean expression with a value from an environment:

```
def subst(m: Map[Int32, Bool], e: Expr[s]): Expr[(s - Var) + Cst] =
    choose* e {
        case Var(x)     => Cst(Map.getWithDefault(x, false, m))
        case Cst(b)     => Cst(b)
        case Not(x)     => Not(subst(m, x))
        case Or(x, y)   => Or(subst(m, x), subst(m, y))
        case And(x, y)  => And(subst(m, x), subst(m, y))
        case Xor(x, y)  => Xor(subst(m, x), subst(m, y))
    }
```

Here the return type of `subst` tells us that the function eliminates the `Var` variant. The typing is perhaps not as precise as we would expect. In particular, the return type also states that the result will contain the `Cst` variant, even if the input contained neither `Var` nor `Cst`.

The original type system, as presented in Madsen et al. [34], did not support subqualification, but relied solely on polymorphism. In that system, `Expr[{Var}]` was incompatible with `Expr[{Var, Cst}]`. As a consequence, every constructor had to be given a polymorphic type scheme (`Var` is used both as a term and a type):

$$\mathtt{Var} : \forall \alpha.\, \mathtt{Int32} \longrightarrow \mathtt{Expr}[\mathtt{Var} + \alpha]$$

With subqualification, a more natural monomorphic type is possible: $\mathtt{Var} : \mathtt{Int32} \longrightarrow \mathtt{Expr}[\mathtt{Var}]$. The lack of subqualification means that a function like:

```
def mkNot(expr: Expr[s]): Expr[Not + s] = Not(expr)
```

does not type check, because `expr` has type $\mathtt{Expr}[s]$ for a fixed s, which is incompatible with the `Not` constructor because it requires `Not` in the index. The type-level indexes are mismatched and there is no subqualification so this function, with the shown signature, is ill-typed.

We now turn to one of our main contributions: a polymorphic type system with qualified types that range over Boolean algebras while supporting subqualification. This system overcomes the limitations of Flix's type and effect system and the limitations of restrictable variants.

s, t	$::=$	**Terms**
$\mid$	n_P	integer values
$\mid$	$\lambda(x)_P.t$	term abstraction
$\mid$	x	term variable
$\mid$	$s(t)$	term application
$\mid$	$\Lambda(X <: S)_P.t$	type abstraction
$\mid$	$\Lambda(Y <: Q)_P.t$	qualifier abstraction
$\mid$	$s[S]$	type application
$\mid$	$s\{\!\{Q\}\!\}$	qualifier application
$\mid$	$\texttt{upqual}\ Q\ s$	qualifier upcast
$\mid$	$\texttt{assert}\ Q\ s$	qualifier assertion
$\mid$	$\texttt{if0}\ s\ \texttt{then}\ t_1\ \texttt{else}\ t_2$	conditional

Γ	$::=$	**Environment**
$\mid$	$\cdot$	empty
$\mid$	$\Gamma,\ x : T$	term binding
$\mid$	$\Gamma,\ X <: S$	type binding
$\mid$	$\Gamma,\ Y <: Q$	qualifier binding

v	$::=$	**Values**
$\mid$	n_P	
$\mid$	$\lambda(x)_P.t$	
$\mid$	$\Lambda(X <: S)_P.t$	
$\mid$	$\Lambda(Y <: Q)_P.t$	

S	$::=$	**Simple Types**
$\mid$	$\top$	top type
$\mid$	$\texttt{int}$	integer type
$\mid$	$T_1 \rightarrow T_2$	function type
$\mid$	X	type variable
$\mid$	$\forall(X <: S).T$	for-all type
$\mid$	$\forall(Y <: Q).T$	qualifier for-all type

T, U	$::=$	**Qualified Types**
$\mid$	$\{Q\}\ S$	qualified type

P, Q, R	$::=$	**Qualifiers**
$\mid$	b	base elements
$\mid$	Y	qualifier variables
$\mid$	$Q \wedge R \mid Q \vee R$	meets and joins
$\mid$	$\sim Q$	negation

C	$::=$	**Concrete Qualifiers**
$\mid$	b	base elements

Fig. 1. The syntax of System $\mathsf{F}_{<:\mathsf{B}}$, as an extension of System $\mathsf{F}_{<:\mathsf{Q}}$ [25]. Changes are highlighted.

3 Calculus

In this section, we first introduce System $\mathsf{F}_{<:\mathsf{B}}$, a typed polymorphic calculus with support for Boolean algebra-based type qualifiers building on the work of Foster et al. [15] and Lee et al. [25]. System $\mathsf{F}_{<:\mathsf{B}}$ serves as a base calculus for providing support for Boolean algebra-based type qualifiers to downstream applications, such as effect exclusion or restrictable variants. System $\mathsf{F}_{<:\mathsf{B}}$ supports the natural subtyping that arises from type qualifiers [15] as well as the expressiveness of Boolean formulas. We show that System $\mathsf{F}_{<:\mathsf{B}}$ satisfies standard soundness properties. We then demonstrate how System $\mathsf{F}_{<:\mathsf{B}}$ can be used in practice by applying it to model effect exclusion with subeffecting in a polymorphic effect calculus System $\mathsf{F}_{<:\mathsf{BE}}$.

3.1 Qualified Types with Boolean Algebras

Now, we proceed by presenting System $\mathsf{F}_{<:\mathsf{B}}$ and showing that standard safety properties hold.

Syntax. The syntax of System $\mathsf{F}_{<:\mathsf{B}}$ is shown in Figure 1. The syntax follows from that of System $\mathsf{F}_{<:\mathsf{Q}}$ in Lee et al. [25], augmenting System $\mathsf{F}_{<:}$ with support for type qualifiers via constructs for annotating qualifiers on values. In addition, to give meaningful runtime semantics, we include (as in Foster et al. [15] and Lee et al. [25]) support for checking and asserting qualifiers (via `assert` and `upqual`). Our major change in System $\mathsf{F}_{<:\mathsf{B}}$ from System $\mathsf{F}_{<:\mathsf{Q}}$ is to draw our base set of qualifiers from a base Boolean algebra B, equipped with distinguished values $\top$ and $\bot$, operations meet ($\sqcap$), join ($\sqcup$), negation ($\neg$), and with ordering $\sqsubseteq$. In addition, we reflect these operations in the syntax of qualifiers with $\wedge$, $\vee$, and $\sim$, respectively.

Evaluation for System $\mathsf{F}_{<:\mathsf{B}}$ $\boxed{s \longrightarrow t}$ and $\boxed{\mathtt{eval}\, Q}$

$$(\lambda(x)_P.t)(v) \longrightarrow t[x \mapsto v] \qquad \text{(BETA-V)}$$

$$(\Lambda(X <: S)_P.t)[S'] \longrightarrow t[X \mapsto S'] \qquad \text{(BETA-T)}$$

$$(\Lambda(Y <: Q)_P.t)\{\!\{Q'\}\!\} \longrightarrow t[Y \mapsto Q'] \qquad \text{(BETA-Q)}$$

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{upqual}\; Q\; v \longrightarrow v \text{ retagged with } Q} \qquad \text{(UPQUAL)}$$

$$\frac{v \text{ tagged with } P \qquad \mathtt{eval}(P) \sqsubseteq \mathtt{eval}(Q)}{\mathtt{assert}\; Q\; v \longrightarrow v} \qquad \text{(ASSERT)}$$

$$\mathtt{if0}\; 0\; \mathtt{then}\; s\; \mathtt{else}\; t \longrightarrow s \qquad \text{(IF-TRUE)}$$

$$\frac{v \neq 0}{\mathtt{if0}\; v\; \mathtt{then}\; s\; \mathtt{else}\; t \longrightarrow t} \qquad \text{(IF-FALSE)}$$

$$\frac{s \longrightarrow t}{E[s] \longrightarrow E[t]} \qquad \text{(CONTEXT)}$$

E	::=	**Evaluation Context**
	|	$[\,]$
	|	$E(t) \mid v(E)$
	|	$E[S] \mid E\{\!\{Q\}\!\}$
	|	$\mathtt{upqual}\; P\; E$
	|	$\mathtt{assert}\; P\; E$
	|	$\mathtt{if0}\; E\; \mathtt{then}\; s\; \mathtt{else}\; t$

$\mathtt{eval}(P)$	::=		**Partial Qualifier Evaluation**
	| C	=>	C
	| $P \wedge R$	=>	$\mathtt{eval}(P) \sqcap \mathtt{eval}(R)$
	| $P \vee R$	=>	$\mathtt{eval}(P) \sqcup \mathtt{eval}(R)$
	| $\sim P$	=>	$\neg\, \mathtt{eval}(P)$

Fig. 2. Reduction rules for System $\mathsf{F}_{<:\mathsf{B}}$, as an extension of System $\mathsf{F}_{<:\mathsf{Q}}$ [25]. Changes are highlighted.

Values and Qualifiers. Now, with our syntax presented, we discuss how we extend values with support for qualifiers. Following Foster et al. [15, Section 2.2] and Lee et al. [25], we *tag* each value with a qualifier expression P denoting the qualifier that the value should be typed at, and we support *asserting* and *upcasting* qualifier tags. For example, the value $\lambda(x : \mathtt{Int})_\top.x$ represents the integer identity function qualified at $\top$. For brevity, we say "v tagged with P" to mean that the runtime tag attached to v is P; here, $\lambda(x : \mathtt{Int})_\top.x$ is tagged with $\top$.

Semantics. The evaluation rules of System $\mathsf{F}_{<:\mathsf{B}}$ in Figure 2 remain largely unchanged from System $\mathsf{F}_{<:\mathsf{Q}}$ and System $\mathsf{F}_{<:}$, except that qualifier evaluation is extended to account for negation. Evaluation converts syntactic algebra operations into the actual underlying algebra operations. Note that while eval is partial and is only defined in terms of concrete qualifier expressions without variables, it will only be used on concrete qualifier expressions during reduction.

Subqualification. System $\mathsf{F}_{<:\mathsf{B}}$ extends System $\mathsf{F}_{<:\mathsf{Q}}$'s subqualification rules to account for distributivity via (SQ-DIST) and negation via (SQ-NEG-INTRO) and (SQ-NEG-ELIM), shown in Figure 3. These additional rules extend the free lattice axioms of System $\mathsf{F}_{<:\mathsf{Q}}$ to the rules necessary to capture the structure of a free Boolean algebra. It is straightforward to show, by induction, that these rules define a Boolean algebra (modulo bounds on free variables), and to show that the resulting Boolean algebra embeds in any other (modulo bounds on free variables). Note that transitivity is now an explicit rule in System $\mathsf{F}_{<:\mathsf{B}}$, as the new Boolean algebra rules do not easily admit integrated transitivity. Also note that only one-sided distributivity is required; the other distributivity rule can be proven from the one given (as we do in our mechanization).

Subqualification for System $F_{<:B}$

$$\boxed{\Gamma \vdash Q <: R}$$

$$\frac{Q \text{ well formed in } \Gamma}{\Gamma \vdash Q <: \top} \quad (\textsc{sq-top})$$

$$\frac{Q \text{ well formed in } \Gamma}{\Gamma \vdash \bot <: Q} \quad (\textsc{sq-bot})$$

$$\frac{\Gamma \vdash Q <: R_1}{\Gamma \vdash Q <: R_1 \vee R_2} \quad (\textsc{sq-join-intro-1})$$

$$\frac{\Gamma \vdash Q <: R_2}{\Gamma \vdash Q <: R_1 \vee R_2} \quad (\textsc{sq-join-intro-2})$$

$$\frac{\Gamma \vdash R_1 <: Q \qquad \Gamma \vdash R_2 <: Q}{\Gamma \vdash R_1 \vee R_2 <: Q} \quad (\textsc{sq-join-elim})$$

$$\Gamma \vdash P \wedge (Q \vee R) <: (P \wedge Q) \vee (P \wedge R) \quad (\textsc{sq-dist})$$

$$\Gamma \vdash \top <: Q \vee (\sim Q) \quad (\textsc{sq-neg-intro})$$

$$\Gamma \vdash Q \wedge (\sim Q) <: \bot \quad (\textsc{sq-neg-elim})$$

$$\frac{\Gamma \vdash Q <: l \qquad \Gamma \vdash l = \mathsf{eval}\, Q' \qquad \Gamma \vdash Q' <: R}{\Gamma \vdash Q <: R} \quad (\textsc{sq-eval-intro})$$

$$\frac{\Gamma \vdash R_1 <: Q}{\Gamma \vdash R_1 \wedge R_2 <: Q} \quad (\textsc{sq-meet-elim-1})$$

$$\frac{\Gamma \vdash R_2 <: Q}{\Gamma \vdash R_1 \wedge R_2 <: Q} \quad (\textsc{sq-meet-elim-2})$$

$$\frac{\Gamma \vdash Q <: R_1 \qquad \Gamma \vdash Q <: R_2}{\Gamma \vdash Q <: R_1 \wedge R_2} \quad (\textsc{sq-meet-intro})$$

$$\frac{Y <: Q \in \Gamma}{\Gamma \vdash Y <: Q} \quad (\textsc{sq-var})$$

$$\Gamma \vdash Q <: Q \quad (\textsc{sq-refl})$$

$$\frac{\Gamma \vdash P <: Q \qquad \Gamma \vdash Q <: R}{\Gamma \vdash P <: R} \quad (\textsc{sq-trans})$$

$$\frac{b_1, b_2 \in B \qquad b_1 \sqsubseteq b_2}{\Gamma \vdash b_1 <: b_2} \quad (\textsc{sq-lift})$$

$$\frac{\Gamma \vdash Q <: Q' \qquad \Gamma \vdash l = \mathsf{eval}\, Q' \qquad \Gamma \vdash l <: R}{\Gamma \vdash Q <: R} \quad (\textsc{sq-eval-elim})$$

Fig. 3. Subqualification rules of System $F_{<:B}$, as an extension of System $F_{<:Q}$ [25]. Changes are highlighted.

Typing for System $F_{<:B}$

$$\boxed{\Gamma \vdash t : T}$$

$$\frac{}{\Gamma \vdash n_P : \{P\}\ \mathsf{Int}} \quad (\textsc{int})$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad (\textsc{var})$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda(x)_P.t : \{P\}\ T_1 \to T_2} \quad (\textsc{abs})$$

$$\frac{\Gamma, X <: S \vdash t : T}{\Gamma \vdash \Lambda(X <: S)_P.t : \{P\}\ \forall(X <: S).T} \quad (\textsc{t-abs})$$

$$\frac{\Gamma, Y <: Q \vdash t : T}{\Gamma \vdash \Lambda(Y <: Q)_P.t : \{P\}\ \forall(Y <: Q).T} \quad (\textsc{q-abs})$$

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathsf{assert}\, P\ t : \{Q\}\ S} \quad (\textsc{typ-assert})$$

$$\frac{\Gamma \vdash t : \{Q\}\ T_1 \to T_2 \qquad \Gamma \vdash s : T_1}{\Gamma \vdash t(s) : T_2} \quad (\textsc{app})$$

$$\frac{\Gamma \vdash t : \{Q\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \vdash t[S'] : T[X \mapsto S']} \quad (\textsc{t-app})$$

$$\frac{\Gamma \vdash t : \{R\}\ \forall(Y <: Q).T \qquad \Gamma \vdash Q' <: Q}{\Gamma \vdash t\{\!\{Q'\}\!\} : T[Y \mapsto Q']} \quad (\textsc{q-app})$$

$$\frac{\Gamma \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash s : T_2} \quad (\textsc{sub})$$

$$\frac{\Gamma \vdash t : \{Q\}\ S \qquad \Gamma \vdash Q <: P}{\Gamma \vdash \mathsf{upqual}\, P\ t : \{P\}\ S} \quad (\textsc{typ-upqual})$$

$$\frac{\Gamma \vdash s : \{P\}\ \mathsf{Int} \qquad \Gamma \vdash t : T \qquad \Gamma \vdash u : T}{\Gamma \vdash \mathtt{if0}\, s\ \mathsf{then}\, t\ \mathsf{else}\, u : T} \quad (\textsc{typ-Ifthenelse})$$

Fig. 4. Typing rules for System $F_{<:B}$ (and of System $F_{<:Q}$).

Subtyping for System $\mathsf{F}_{<:\mathsf{B}}$

$\boxed{\Gamma \vdash S_1 <: S_1 \text{ and } \Gamma \vdash T_1 <: T_2}$

$$\Gamma \vdash S <: \top \qquad \text{(SUB-TOP)}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X <: X} \qquad \text{(SUB-REFL-SVAR)}$$

$$\frac{X <: S_1 \in \Gamma \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash X <: S_2} \qquad \text{(SUB-SVAR)}$$

$$\frac{\Gamma \vdash Q_1 <: Q_2 \qquad \Gamma \vdash S_1 <: S_2}{\Gamma \vdash \{Q_1\}\ S_1 <: \{Q_2\}\ S_2} \qquad \text{(SUB-QTYPE)}$$

$$\frac{\Gamma \vdash T_2 <: T_1 \qquad \Gamma \vdash T_3 <: T_4}{\Gamma \vdash T_1 \to T_3 <: T_2 \to T_4} \qquad \text{(SUB-ARROW)}$$

$$\frac{\Gamma \vdash S_2 <: S_1 \qquad \Gamma, X <: S_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(X <: S_1).T_1 <: \forall(X <: S_2).T_2} \qquad \text{(SUB-ALL)}$$

$$\frac{\Gamma \vdash Q_2 <: Q_1 \qquad \Gamma, Y <: Q_2 \vdash T_1 <: T_2}{\Gamma \vdash \forall(Y <: Q_1).T_1 <: \forall(Y <: Q_2).T_2} \qquad \text{(SUB-QALL)}$$

Fig. 5. Subtyping rules of System $\mathsf{F}_{<:\mathsf{B}}$ (and of System $\mathsf{F}_{<:\mathsf{Q}}$).

Subtyping and Typing. Finally, System $\mathsf{F}_{<:\mathsf{B}}$ inherits all of System $\mathsf{F}_{<:\mathsf{Q}}$'s subtyping and typing rules; all of its changes are in the structure and semantics of its type qualifiers, not in the structure of its types or terms. For completeness, we reproduce these rules in Figures 4 and 5.

3.2 Metatheory

System $\mathsf{F}_{<:\mathsf{B}}$ satisfies the standard progress and preservation theorems.

Theorem 3.1 (Preservation). *Suppose* $\Gamma \vdash s : T$, *and* $s \longrightarrow t$. *Then* $\Gamma \vdash t : T$ *as well.*

Theorem 3.2 (Progress). *Suppose* $\varnothing \vdash s : T$. *Then either* s *is a value, or* $s \longrightarrow t$ *for some term* t.

While System $\mathsf{F}_{<:\mathsf{B}}$ does not place any interpretation on qualifiers outside of `upqual` and `assert`, such a system can already be useful. For one, the static type of a value will always be greater than the tag annotated on it, and that correspondence is preserved through reduction. This property can be used to enforce safety constraints, as Foster et al. [15] and Lee et al. [25] note.

3.3 Applying System $\mathsf{F}_{<:\mathsf{B}}$

We can use System $\mathsf{F}_{<:\mathsf{B}}$ as a framework for type systems with qualifiers that are Boolean algebras. To instantiate the framework, follow the four steps:

(1) **Select an algebra:** Choose the base Boolean algebra B that qualifiers will be drawn from. For a simple nullability system, B can simply be the two element Boolean algebra $\{\top, \bot\}$, where $\top$ means nullable and $\bot$ means not-null.
(2) **Define value qualifiers:** Determine how values will be qualified by the typing rules. For example, in a nullability system, null will be qualified with $\top$ while references that are not null will be qualified with $\bot$.
(3) **Define qualifier elimination:** Determine how to type elimination forms (e.g function application or other expressions which consume values) with qualifiers. For example, in a nullability system, the typing rule for dereference would require that the qualifier of that reference is $\bot$.
(4) **Define operational semantics (Optional):** Augment the operational semantics to store and check qualifiers in order to catch undesirable operations at runtime. For example, in a nullability system, dereferencing a reference qualified with $\top$ would get stuck. Now if one can prove that progress and preservation – soundness – hold for the resulting system, then this guarantees that operations that are not allowed by the qualifiers will not be performed.

Here, for example, one can show that nullable references will never be dereferenced. Since these checks only stop programs from evaluating, qualifiers and checks can be erased safely.

3.4 Application: Polymorphic Effect Exclusion

With our base rules defined in System $\mathsf{F}_{<:\mathsf{B}}$, we now show how they can be remixed as a design pattern to support applications such as effect exclusion. We thus apply it to model an effect safety system with support for *effect exclusion*, using it as a basis of an effect calculus System $\mathsf{F}_{<:\mathsf{BE}}$. System $\mathsf{F}_{<:\mathsf{BE}}$ itself is based on an earlier calculus System $\mathsf{F}_{<:\mathsf{QC}}$ by Lee et al. [25] for a simpler function coloring system, but we extend it on top of System $\mathsf{F}_{<:\mathsf{B}}$ to support effect exclusion.

Using System $\mathsf{F}_{<:\mathsf{B}}$ *as a design recipe for System* $\mathsf{F}_{<:\mathsf{BE}}$. We illustrate how System $\mathsf{F}_{<:\mathsf{B}}$ can be used as a design recipe for building System $\mathsf{F}_{<:\mathsf{BE}}$.

1. **Select an algebra:** For System $\mathsf{F}_{<:\mathsf{BE}}$, we select the base Boolean algebra B to be the subset algebra of effects. For example, if we have the `IO` and `Block` effects, then the subset algebra consists of `{}`, `{IO}`, `{Block}`, and `{IO, Block}`.
2. **Define value qualifiers:** In System $\mathsf{F}_{<:\mathsf{BE}}$, effectful values – term and type abstractions – are qualified with the set of effects they are allowed to perform. For example, a term abstraction that uses a primitive `IO` effect, like printing to the screen, will be qualified with `{IO}`. To model performing primitive effects in a calculus, we introduce a $\mathsf{do}\; b$ expression, which performs the effect labelled by b. The expression $\mathsf{do}\; b$ simply returns a value immediately after checking that the effect b is allowed by the current evaluation context; in a real implementation, an effectful operation would also be performed.
3. **Define qualifier elimination:** Elimination forms, like term or type application, will ensure that the qualifier of the function being applied is included in the set of qualifiers (effects) allowed by the enclosing context.
4. **Define operational semantics (Optional):** We equip System $\mathsf{F}_{<:\mathsf{BE}}$ with operational semantics that ensure that effects can only be performed in contexts where they are allowed. This is shown in Section 3.4.

Syntax. Figure 6 presents the additional syntax of System $\mathsf{F}_{<:\mathsf{BE}}$ with support for performing effectful operations and excluding effects from terms. Qualifier annotations, as noted above, are reused on abstraction terms to denote the set of possible effects an abstraction may perform.

Evaluation. To model effect safety, Figure 7 describes the operational semantics of System $\mathsf{F}_{<:\mathsf{BE}}$ using Felleisen and Friedman [14] -style CK semantics, extended with special *barrier* frames installed on the stack denoting the effect of the function that was called, and *fences* excluding effects from evaluation contexts. When a function is called, we place a *barrier* with the evaluated effect set of the function itself; so a term like

$$\langle 1, \mathsf{app}\, \lambda(x)_{\bot}.x :: \kappa\rangle \quad\longrightarrow\quad \langle 1, \mathtt{barrier}\, \bot :: \kappa\rangle$$

placing a barrier marking a function allowing only the $\bot$ effect above it in the continuation stack.

Barriers are used to ensure that applied functions are compatible with the rest of a stack. For example, the following term places incompatible barriers on the stack.

$$\begin{aligned} \langle (\lambda(x)_{\top}.t)\; v, \mathtt{barrier}\, \bot\rangle &\longrightarrow \langle \lambda(x)_{\top}.t, \mathsf{arg}\, v :: \mathtt{barrier}\, \bot\rangle \\ &\longrightarrow \langle v, \mathsf{app}\, \lambda(x)_{\top}.t :: \mathtt{barrier}\, \bot\rangle \\ &\longrightarrow \text{gets stuck.} \end{aligned}$$

To exclude effects, $\mathtt{fence}\, b$ frames are used to mark the extent of an evaluation context where the effects labelled by b cannot be performed. Placing a barrier frame $\mathtt{barrier}\, b$ on top of a $\mathtt{fence}\, d$

t, c	$::=$		**Terms**
	$\ldots$		as before, except:
	$\mid$	$\mathtt{without}\, C\; t$	exclusion fence
	$\mid$	$\mathtt{do}\;\, b$	effectful operation
κ	$::=$		**Evaluation Context**
	$\mid$	$[\,]$	
	$\mid$	$f :: \kappa$	

f	$::=$		**Evaluation Frames**
	$\mid$	$\mathtt{barrier}\, C$	barrier
	$\mid$	$\mathtt{fence}\, C$	fence
	$\mid$	$\mathsf{arg}\, t$	argument
	$\mid$	$\mathsf{app}\, v$	application
	$\mid$	$\mathtt{targ}\, T$	type application
	$\mid$	$\mathtt{qarg}\, Q$	qualifier application
	$\mid$	$\mathtt{if0}\, s\; t$	if

Fig. 6. The syntax of System $\mathsf{F}_{<:\mathsf{BE}}$, based off of System $\mathsf{F}_{<:\mathsf{QC}}$ [25]. Non-standard context frames highlighted.

Evaluation for System $\mathsf{F}_{<:\mathsf{BE}}$

$$\boxed{\langle s, \kappa \rangle \longrightarrow \langle t, \kappa' \rangle}$$

$$\langle s(t), \kappa \rangle \longrightarrow \langle s, \mathsf{arg}\, t :: \kappa \rangle \quad (\textsc{cong-app})$$

$$\langle v, \mathsf{arg}\, t :: \kappa \rangle \longrightarrow \langle t, \mathsf{app}\, v :: \kappa \rangle \quad (\textsc{cong-arg})$$

$$\langle s[S], \kappa \rangle \longrightarrow \langle s, \mathtt{targ}\, S :: \kappa \rangle \quad (\textsc{cong-tapp})$$

$$\langle s\{\!\{Q\}\!\}, \kappa \rangle \longrightarrow \langle s, \mathtt{qarg}\, Q :: \kappa \rangle \quad (\textsc{cong-qapp})$$

$$\langle v, \mathtt{barrier}\, C :: \kappa \rangle \longrightarrow \langle v, \kappa \rangle \quad (\textsc{break-barrier})$$

$$\langle v, \mathtt{fence}\, C :: \kappa \rangle \longrightarrow \langle v, \kappa \rangle \quad (\textsc{jump-fence})$$

$$\langle \mathtt{without}\, C\; t, \kappa \rangle \longrightarrow \langle t, \mathtt{fence}\, C :: \kappa \rangle \quad (\textsc{reduce-fence})$$

$$\langle \mathtt{if0}\, s\; \mathsf{then}\; t_1\; \mathsf{else}\; t_2, \kappa \rangle \longrightarrow \langle s, \mathtt{if0}\, t_1\; t_2 :: \kappa \rangle \quad (\textsc{cong-if})$$

$$\langle 0, \mathtt{if0}\, s\; t :: \kappa \rangle \longrightarrow \langle s, \kappa \rangle \quad (\textsc{reduce-iftrue})$$

$$\frac{v \neq 0}{\langle v, \mathtt{if0}\, s\; t :: \kappa \rangle \longrightarrow \langle t, \kappa \rangle} \quad (\textsc{reduce-iffalse})$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\;\, C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\;\, C_i \text{ frames on } \kappa \end{array}}{\langle v, \mathsf{app}\, \lambda(x)_P.t :: \kappa \rangle \longrightarrow \langle t[x \mapsto v], \mathtt{barrier}\, C :: \kappa \rangle} \quad (\textsc{reduce-app})$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\;\, C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\;\, C_i \text{ frames on } \kappa \end{array}}{\langle \Lambda(X <: S)_P.t, \mathtt{targ}\, S' :: \kappa \rangle \longrightarrow \langle t[X \mapsto S'], \mathtt{barrier}\, C :: \kappa \rangle} \quad (\textsc{reduce-tapp})$$

$$\frac{\begin{array}{c} C \sqsubseteq C_i \text{ for all } \mathtt{barrier}\;\, C_i \text{ frames on } \kappa \qquad \mathsf{eval}\, P = C \\ C \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\;\, C_i \text{ frames on } \kappa \end{array}}{\langle \Lambda(Y <: Q)_P.t, \mathtt{qarg}\, Q' :: \kappa \rangle \longrightarrow \langle t[Y \mapsto Q'], \mathtt{barrier}\, C :: \kappa \rangle} \quad (\textsc{reduce-qapp})$$

$$\frac{\begin{array}{c} b \sqsubseteq C_i \text{ for all } \mathtt{barrier}\;\, C_i \text{ frames on } \kappa \\ b \sqcap C_i \sqsubseteq \bot \text{ for all } \mathtt{fence}\;\, C_i \text{ frames on } \kappa \end{array}}{\langle \mathsf{do}\, b, \kappa \rangle \longrightarrow \langle 1, \kappa \rangle} \quad (\textsc{reduce-do})$$

Fig. 7. Operational Semantics (CK-style) for System $\mathsf{F}_{<:\mathsf{BE}}$ (based off of System $\mathsf{F}_{<:\mathsf{QC}}$ [25])

frame fails unless $b \sqsubseteq \neg d$. Finally, effects can only be performed by $\mathsf{do}\, b$ if b is compatible – contained within – all `barrier` frames on the stack and disjoint from all `fence` frames on the stack.

Typing. To guarantee soundness, Figure 8 endows the typing rules of System $\mathsf{F}_{<:\mathsf{BE}}$ with modified rules for keeping track of the effect that context abstractions need. We extend the typing rules with an effect context R to keep track of the effects a function is allowed to perform. This effect context R is simply a qualifier expression, and is introduced by the rules for typing abstractions by lifting the qualifier tagged on those abstractions – see rules (A-abs), (A-t-abs), and (A-q-abs).

To ensure effect safety, both the elimination forms for abstraction application and effect application check the effect context to ensure that the effects of the operation or function invocation are compatible with the effects allowed by the current context.

Context and Configuration Typing. Progress and preservation theorems are traditionally stated in terms of the behaviour of a single step reduction relation on terms.

Typing for System $\mathsf{F}_{<:\mathsf{BE}}$

$$\boxed{\Gamma \mid R \vdash s : T}$$

$$\frac{x : T \in \Gamma}{\Gamma \mid R \vdash x : T} \quad (\text{A-VAR})$$

$$\frac{\Gamma \mid R \vdash t : \{R\}\ T_1 \to T_2 \qquad \Gamma \mid R \vdash s : T_1}{\Gamma \mid R \vdash t(s) : T_2} \quad (\text{A-APP})$$

$$\frac{\Gamma, x : T_1 \mid P \vdash t : T_2}{\Gamma \mid \bot \vdash \lambda(x)_P.t : \{P\}\ T_1 \to T_2} \quad (\text{A-ABS})$$

$$\frac{\Gamma \mid R \vdash t : \{R\}\ \forall(X <: S).T \qquad \Gamma \vdash S' <: S}{\Gamma \mid R \vdash t[S'] : T[X \mapsto S']} \quad (\text{A-T-APP})$$

$$\frac{\Gamma, X <: S \mid P \vdash t : T}{\Gamma \mid \bot \vdash \Lambda(X <: S)_P.t : \{P\}\ \forall(X <: S).T} \quad (\text{A-T-ABS})$$

$$\frac{\Gamma \mid R \vdash t : \{R\}\ \forall(Y <: Q).T \qquad \Gamma \vdash Q' <: Q}{\Gamma \mid R \vdash t\{\!\{Q'\}\!\} : T[Y \mapsto Q']} \quad (\text{A-Q-APP})$$

$$\frac{\Gamma, Y <: Q \mid P \vdash t : T}{\Gamma \mid \bot \vdash \Lambda(Y <: Q)_P.t : \{P\}\ \forall(Y <: Q).T} \quad (\text{A-Q-ABS})$$

$$\Gamma \mid b \vdash \texttt{do}\, b : \{\bot\}\ \texttt{int} \quad (\text{A-DO})$$

$$\frac{\Gamma \vdash R <: \ \sim b \qquad \Gamma \mid R \vdash s : T}{\Gamma \mid R \vdash \texttt{without}\, b\ s : T} \quad (\text{A-WITHOUT})$$

$$\frac{\Gamma \mid R \vdash s : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \mid R \vdash s : T_2} \quad (\text{A-SUB})$$

$$\frac{\Gamma \mid R \vdash s : \{P\}\ \texttt{int} \qquad \Gamma \mid R \vdash t_1 : T \qquad \Gamma \mid R \vdash t_2 : T}{\Gamma \mid R \vdash \texttt{if0}\, s\ \texttt{then}\, t_1\ \texttt{else}\, t_2 : T} \quad (\text{A-IF})$$

$$\frac{\Gamma \mid R \vdash s : T \qquad \Gamma \vdash R <: Q}{\Gamma \mid Q \vdash s : T} \quad (\text{A-SUB-EFF})$$

Fig. 8. Typing rules for System $\mathsf{F}_{<:\mathsf{BE}}$ based on System $\mathsf{F}_{<:\mathsf{QC}}$ [25]. Changes from System $\mathsf{F}_{<:\mathsf{B}}$ are highlighted.

Metatheorem 1 (Progress). *If $\varnothing \vdash e : U$, then either e is a value or there exists a term t such that $s \longrightarrow t$.*

Metatheorem 2 (Preservation). *If $E \vdash e : U$ and $e \longrightarrow t$, then $E \vdash t : U$.*

Our operational semantics for System $\mathsf{F}_{<:\mathsf{BE}}$ are based on the CK machine [14], which operates on an explicit machine configuration consisting of a term s in redex position and an evaluation context κ around s. In this setting, our original program s can be recovered by filling in the hole in κ with s. So we need to define what it means to give a type to $\kappa[s]$.

This we do in two parts. First, we define a context typing relation $E \mid R \vdash \kappa : T \rightsquigarrow U$ that describes how a context κ expects a type T and produces a type U. Second, we define a machine typing relation $E \mid R \vdash \langle s, \kappa \rangle : U$, which defines how a particular machine configuration $\langle s, \kappa \rangle$ should produce U when evaluated, and which captures the intuitive[1] notion that $\kappa[s]$ should type to U. Here, κ should type to $T \rightsquigarrow U$ and s should type to T. These rules are defined in Figure 9.

Metatheory. With all this, we can state and prove progress and preservation for System $\mathsf{F}_{<:\mathsf{BE}}$.

Theorem 3.3 (Progress of System $\mathsf{F}_{<:\mathsf{BE}}$). *Suppose $\langle s, \kappa \rangle$ is a well-typed (in an empty environment) machine configuration: namely, $\varnothing \mid R \vdash \langle s, \kappa \rangle : U$. Then either s is a value and k the empty continuation, or there is a machine state $\langle t, \kappa' \rangle$ that it steps to.*

Theorem 3.4 (Preservation of System $\mathsf{F}_{<:\mathsf{BE}}$). *Suppose $\langle s, \kappa \rangle$ is a well-typed machine configuration, namely: $E \mid R \vdash \langle s, \kappa \rangle : U$. If it steps to another configuration $\langle t, \kappa' \rangle$, then $E \mid R \vdash \langle t, \kappa' \rangle : U$ as well.*

[1]This can be formalized by defining the appropriate notion of *plugging* a term into a context. For an example, see the proof artifact of [9].

Configuration Typing for System $\mathsf{F}_{<:\mathsf{BE}}$ $\boxed{E \mid R \vdash \kappa : T \rightsquigarrow U}$ and $\boxed{E \mid R \vdash \langle s, \kappa \rangle : U}$

$$\frac{}{E \mid R \vdash [\,] : T \rightsquigarrow T}\ (\textsc{ctx-empty})$$

$$\frac{E \mid R \vdash \kappa : T_2[X \mapsto T_1'] \rightsquigarrow U \qquad E \vdash T_1' <: T_1}{E \mid R \vdash (\mathtt{targ}\, T_1' :: \kappa) : \{R\}\ \forall(X <: T_1).T_2 \rightsquigarrow U}\ (\textsc{ctx-tabs})$$

$$\frac{E \mid R \vdash \kappa : T_2 \rightsquigarrow U \qquad E \mid R \vdash e_1 : T_1}{E \mid R \vdash (\mathtt{arg}\, e_1 :: \kappa) : \{R\}\ T_1 \to T_2 \rightsquigarrow U}\ (\textsc{ctx-abs})$$

$$\frac{E \mid R \vdash \kappa : T_2[X \mapsto Q'] \rightsquigarrow U \qquad E \vdash Q' <: Q}{E \mid R \vdash (\mathtt{qarg}\, Q' :: \kappa) : \{R\}\ \forall(X <: Q).T_2 \rightsquigarrow U}\ (\textsc{ctx-qabs})$$

$$\frac{E \mid R \vdash \kappa : T_2 \rightsquigarrow U \qquad E \mid R \vdash v : \{R\}\ T_1 \to T_2}{E \mid R \vdash (\mathtt{app}\, v :: \kappa) : T_1 \rightsquigarrow U}\ (\textsc{ctx-app})$$

$$\frac{E \mid R \vdash \kappa : T \rightsquigarrow U}{E \mid (R \wedge \neg b) \vdash (\mathtt{fence}\, b :: \kappa) : T \rightsquigarrow U}\ (\textsc{ctx-without})$$

$$\frac{E \mid R \vdash \kappa : T \rightsquigarrow U \qquad \mathtt{eval}\, R = b_2 \qquad b_1 \sqsubseteq b_2}{E \mid b_1 \vdash (\mathtt{barrier}\, b_1 :: \kappa) : T \rightsquigarrow U}\ (\textsc{ctx-barrier})$$

$$\frac{E \mid R_2 \vdash \kappa : T \rightsquigarrow U \qquad E \vdash R_1 <: R_2}{E \mid R_1 \vdash \kappa : T \rightsquigarrow U}\ (\textsc{ctx-subeff})$$

$$\frac{E \mid R \vdash \kappa : T_2 \rightsquigarrow U \qquad E \vdash T_1 <: T_2}{E \mid R \vdash \kappa : T_1 \rightsquigarrow U}\ (\textsc{ctx-sub})$$

$$\frac{E \mid R \vdash \kappa : T \rightsquigarrow U \qquad E \mid R \vdash e_1 : T \qquad E \mid R \vdash e_2 : T}{E \mid R \vdash (\mathtt{if0}\, e_1 e_2 :: \kappa) : \{P\}\ \mathtt{int} \rightsquigarrow U}\ (\textsc{ctx-if})$$

$$\frac{E \mid R \vdash \kappa : T \rightsquigarrow U \qquad E \mid R \vdash s : T}{E \mid R \vdash \langle s, \kappa \rangle : U}\ (\textsc{cfg-typed})$$

Fig. 9. Context and Configuration Typing for System $\mathsf{F}_{<:\mathsf{BE}}$.

Note that progress and preservation guarantee meaningful safety properties about System $\mathsf{F}_{<:\mathsf{BE}}$; effectful operations can only be invoked in contexts where they are allowed.

Mechanization. The proofs of the standard soundness theorems of our calculi System $\mathsf{F}_{<:\mathsf{B}}$ and System $\mathsf{F}_{<:\mathsf{BE}}$ have been fully mechanized in Rocq, with inspiration drawn from the mechanization of System $\mathsf{F}_{<:\mathsf{Q}}$ and its derived calculi by Lee et al. [25] and from the mechanization of System $\mathsf{F}_{<:}$ by Aydemir et al. [1].

With System $\mathsf{F}_{<:\mathsf{B}}$ and System $\mathsf{F}_{<:\mathsf{BE}}$ constructed, we show how System $\mathsf{F}_{<:\mathsf{BE}}$ can be applied in practice by using it to add subeffecting to Flix's effect system.

4 The Flix Type and Effect System

Flix is a functional, imperative, and logic programming language with algebraic data types, extensible records, higher-order functions, parametric polymorphism, type classes with higher-kinded types, first-class Datalog constraints, and a polymorphic type and effect system [33, 35] with support for effect handlers, associated effects [29], and polymorphic effect exclusion [30]. The Flix compiler, including the standard library and tests, is approximately 270,000 lines of code. Flix comes with a standard library that is subject to its type and effect system. The library is extensive, offering more than 3,500 public functions and spanning more than 37,000 lines of code.

The Flix type and effect system lacks subeffecting; instead programmers have to manually insert explicit effect upcasts. To overcome this issue, we add a limited form of subeffecting, dubbed *abstraction-site subeffecting*, to the $\lambda_{\complement}$ calculus from Lutze et al. [30]. We present a new calculus $\lambda_{\textsc{flx}}$ to formally describe abstraction-site subeffecting and show that it is an instance of the more general language, System $\mathsf{F}_{<:\mathsf{BE}}$. We then implement abstraction-site subeffecting in the Flix compiler and empirically evaluate its usefulness.

Abstraction-site Subeffecting. Contrary to System $\mathsf{F}_{<:\mathsf{BE}}$, we allow the use of the subeffect rule only in combination with the abstraction rule ($\textsc{F-Abs}$). This idea is known (e.g. by Talpin and

s, t, u	$::=$		**Terms**
	$\mid$	$0, 1, 2, \dots$	integer terms
	$\mid$	$\lambda(x).t$	term abstraction
	$\mid$	x	term variable
	$\mid$	$s(t)$	application
	$\mid$	$\mathtt{let}\, x = s\, \mathtt{in}\, t$	let bind
	$\mid$	$\mathtt{def}\, x = v\, \mathtt{in}\, t$	definition
	$\mid$	$\mathtt{do}\, b$	effect invocation
	$\mid$	$t\, \mathtt{without}\, b$	effect exclusion
	$\mid$	$\mathtt{if0}\, s\, \mathtt{then}\, t\, \mathtt{else}\, u$	conditional
v	$::=$		**Values**
	$\mid$	$0, 1, 2, \dots$	
	$\mid$	$\lambda(x).t$	

T	$::=$		**Types**
	$\mid$	$T_1 \xrightarrow{Q} T_2$	function type
	$\mid$	$\mathtt{int}$	integer
	$\mid$	X	type variable
P, Q, R	$::=$		**Effects**
	$\mid$	$\bot$	no effects
	$\mid$	b	base effects
	$\mid$	Y	effect variables
	$\mid$	$\sim Q$	effect negation
	$\mid$	$Q \wedge R$	effect meet
	$\mid$	$Q \vee R$	effect join
S	$::=$		**Schemes**
	$\mid$	$\forall \overline{XY}.T$	scheme

Fig. 10. Syntax rules for λ_{FLX} taken from $\lambda_{\complement}$ [30] with some mild changes to allow clearer comparison. Note that though the top effect is not present in the syntax, it can be expressed with $\sim \bot$ for example. The $\lambda_{\complement}$ language requires a stricter syntax than shown here, application is $v_1(v_2)$ for example. We permit expressions everywhere, with their obvious desugaring left implicit.

Jouvelot [47]) but has, to our knowledge, not been given a name, so we will call it abstraction-site subeffecting. Abstraction-site subeffecting is simple to understand, easy to implement, and has a low performance impact, even with a generic Boolean unification algorithm. One might question the expressiveness of this limited form of subeffecting, but as we show in Section 5, it is sufficiently expressive to remove all effect casts in the Flix Standard Library.

An alternative to abstraction-site subeffecting would be to inline the subeffecting rule into all typing rules. This would increase the number of variables by orders of magnitude, which then exponentially impacts the performance of unification. This is one reason that abstraction-site subeffecting is an attractive choice.

Flix, with abstraction-site subeffecting, is open source, ready for use, and freely available at

https://flix.dev/ and https://github.com/flix/flix

4.1 Formalization of λ_{FLX}

We present the syntax and type system of λ_{FLX}, which is based on System $\mathsf{F}_{<:\mathsf{BE}}$ and Lutze et al. [30].

Syntax. The syntax of λ_{FLX} is shown in Figure 10. As subeffecting does not change the syntax of the language, there are no syntax changes from $\lambda_{\complement}$ except for superficial ones.

Subeffecting and Typing. As discussed earlier, for practical reasons, we added subeffecting just at abstraction sites. This is implemented by the changes in the (F-ABS) abstraction rule in Figure 11; the effect of the body P can be smaller (subqualified) than the effect wanted on the abstraction Q. As λ_{FLX} has polymorphism but not bounded polymorphism, for brevity we omit the typing context for subqualification ($P <: Q$) in the typing rules; all qualifier (and type) variables have $\top$ as their upper bound. Other than that, the subeffecting rules for λ_{FLX} are the same as the subeffecting rules for System $\mathsf{F}_{<:\mathsf{B}}$ (and System $\mathsf{F}_{<:\mathsf{BE}}$) from Figure 3.

As λ_{FLX} does not have subtyping outside of abstraction-site subeffecting, it has a rule (F-EQ) to handle equivalent but distinct types. This is done via Boolean equivalence ($\equiv_{\mathbb{B}}$), which is structurally defined on the syntax of types with the exception of the effect qualifiers. Effects are tested for

Typing for λ_{FLX} $\boxed{\Gamma \vdash s : T \mid Q}$

$$\frac{i \in \{0, 1, 2, \dots\}}{\Gamma \vdash i : \texttt{int} \mid \bot} \text{ (F-INT)}$$

$$\frac{x : \forall \overline{X}\overline{Y}.T' \in \Gamma \qquad T = T'[\overline{X} \mapsto \overline{T}][\overline{Y} \mapsto \overline{Q}]}{\Gamma \vdash x : T \mid \bot} \text{ (F-VAR)}$$

$$\frac{\Gamma \vdash s : T_1 \xrightarrow{P} T_2 \mid Q \qquad \Gamma \vdash t : T_1 \mid R}{\Gamma \vdash s(t) : T_2 \mid P \vee Q \vee R} \text{ (F-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash t : T_2 \mid P \qquad P <: Q}{\Gamma \vdash \lambda(x).t : T_1 \xrightarrow{Q} T_2 \mid \bot} \text{ (F-ABS)}$$

$$\Gamma \vdash \texttt{do}\, b : \texttt{int} \mid b \text{ (F-DO)}$$

$$\frac{\Gamma \vdash t : T \mid P \qquad P \wedge b \equiv_{\mathbb{B}} \bot}{\Gamma \vdash t\, \texttt{without}\, b : T \mid P} \text{ (F-WITHOUT)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : T_1 \mid \bot \qquad \Gamma, x : S \vdash t : T_2 \mid P \\ S = \forall \overline{X}\overline{Y}.T_1, \\ \dots\text{binding variables in } T_1\text{, not free in } \Gamma\end{array}}{\Gamma \vdash \texttt{def}\, x = v\, \texttt{in}\, t : T_2 \mid P} \text{ (F-DEF)}$$

$$\frac{\Gamma \vdash t : T_1 \mid P \qquad P \equiv_{\mathbb{B}} Q \qquad T_1 \equiv_{\mathbb{B}} T_2}{\Gamma \vdash t : T_2 \mid Q} \text{ (F-EQ)}$$

$$\frac{\Gamma \vdash s : T_1 \mid P \qquad \Gamma, x : T_1 \vdash t : T_2 \mid Q}{\Gamma \vdash \texttt{let}\, x = s\, \texttt{in}\, t : T_2 \mid P \vee Q} \text{ (F-LET)}$$

$$\frac{\Gamma \vdash s : \texttt{int} \mid P \qquad \Gamma \vdash t : T \mid Q \qquad \Gamma \vdash u : T \mid R}{\Gamma \vdash \texttt{if0}\, s\, \texttt{then}\, t\, \texttt{else}\, u : T \mid P \vee Q \vee R} \text{ (F-IF)}$$

Fig. 11. Typing rules for λ_{FLX} based on $\lambda_{\mathbb{C}}$ [30]. Changes are highlighted.

equality by a two-sided subeffecting check. Again, we omit the environment of subeffecting for brevity as all effect variables have $\top$ as their upper bound.

4.2 Soundness of λ_{FLX}

To indicate that the type system of λ_{FLX} is sound, we now give a translation sketch from typings of λ_{FLX} to typings of System $\mathsf{F}_{<:\mathsf{BE}}$, as illustrated in Section 4.2.1.

This translation is mostly straightforward, as λ_{FLX}'s type system can be viewed as the ML-style polymorphic lambda calculus analogue of System $\mathsf{F}_{<:\mathsf{BE}}$. One technical issue is that type variables range over *unqualified* types in System $\mathsf{F}_{<:\mathsf{BE}}$ but over *fully-qualified* types in λ_{FLX}. However, as type variables in λ_{FLX} cannot be further re-qualified, we can simply view a type variable as a pair of a qualifier variable and unqualified type variable in System $\mathsf{F}_{<:\mathsf{BE}}$, as discussed in Lee et al. [25]. This distinction is sometimes shown when grammatically important – for example splitting X into respective type and qualified type part, X_{t} and X_{q} – but it is otherwise left implicit.

The point of this translation is to show that λ_{FLX} is merely a simple and local restriction of System $\mathsf{F}_{<:\mathsf{BE}}$ that maintains soundness. The upside is that this restricted system allows type inference.

4.2.1 *Translation from λ_{FLX} to System $\mathsf{F}_{<:\mathsf{BE}}$.*

Typing rules from λ_{FLX} are shown on the left and the corresponding combined typing rules of System $\mathsf{F}_{<:\mathsf{BE}}$ are shown on the right.

$$\frac{x : \forall \overline{X}\overline{Y}.T' \in \Gamma \qquad T = T'[\overline{X} \mapsto \overline{T}][\overline{Y} \mapsto \overline{Q}]}{\Gamma \vdash x : T \mid \bot} \qquad \frac{\begin{array}{c} x : \forall(\overline{X} <: \top)(\overline{Y} <: \top).T' \in \Gamma \\ T = T'[\overline{X} \to \overline{T}][\overline{Y} \to \overline{Q}]\end{array}}{\Gamma \mid \bot \vdash x[\overline{T}_{\mathsf{t}}]\{\!\{\overline{T}_{\mathsf{q}}, \overline{Q}\}\!\} : T}$$

Variables in λ_{FLX} may be mapped to a type scheme. If that is so, we translate that typing judgment to an instantiated polymorphic function type in System $\mathsf{F}_{<:\mathsf{BE}}$, taking care to split type variables in λ_{FLX} to a pair of unqualified type and qualifier variables in System $\mathsf{F}_{<:\mathsf{BE}}$.

$$\dfrac{\Gamma, x : T_1 \vdash t : T_2 \shortmid P \qquad P <: Q}{\Gamma \vdash \lambda(x).t : T_1 \xrightarrow{Q} T_2 \shortmid \bot} \qquad\qquad \dfrac{\Gamma, x : T_1 \mid P \vdash t : T_2 \qquad \Gamma \vdash P <: Q}{\Gamma \mid \bot \vdash \lambda(x)_P.t : \{Q\}\ T_1 \to T_2}$$

The abstraction typing rule in λ_{FLX} lifts directly to a combination of the abstraction typing rule in System $\mathsf{F}_{<:\mathsf{BE}}$ followed by a subsequent subsumption by (A-SUB).

$$\dfrac{\begin{array}{c}\Gamma \vdash v : T_1 \shortmid \bot \qquad \Gamma, x : S \vdash t : T_2 \shortmid P \\ S = \forall \overline{XY}.T_1, \\ \text{binding variables in } T_1\text{, not free in } \Gamma\end{array}}{\Gamma \vdash \mathtt{def}\ x = v\ \mathtt{in}\ t : T_2 \shortmid P} \qquad \dfrac{\begin{array}{c}\Gamma, \overline{X} <: \top, \overline{Y} <: \top \mid \bot \vdash v : T_1 \\ \Gamma, x : \forall(\overline{XY} <: \top).T_1 \mid P \vdash t : T_2\end{array}}{\Gamma \mid P \vdash (\lambda(x)_P.t)(\Lambda(\overline{X}_{\mathrm{t}}\overline{X}_{\mathrm{q}}\overline{Y} <: \top)_\bot.v) : T_2}$$

Definitions in λ_{FLX} lift to effect-free curried polymorphic functions in System $\mathsf{F}_{<:\mathsf{BE}}$ (noting that v being a value has no side effect when evaluated).

$$\dfrac{\Gamma \vdash s : T_1 \shortmid P \qquad \Gamma, x : T_1 \vdash t : T_2 \shortmid Q}{\Gamma \vdash \mathtt{let}\ x = s\ \mathtt{in}\ t : T_2 \shortmid P \lor Q} \qquad \dfrac{\begin{array}{c}\Gamma \mid P \vdash s : T_1 \qquad \Gamma, x : T_1 \mid Q \vdash t : T_2 \\ \Gamma \vdash P <: P \lor Q \qquad \Gamma \vdash Q <: P \lor Q\end{array}}{\Gamma \mid P \lor Q \vdash (\lambda(x)_Q.t)(s) : T_2}$$

Let binders are converted into an effect-annotated application along with subeffecting.

$$\dfrac{\Gamma \vdash s : T_1 \xrightarrow{P} T_2 \shortmid Q \qquad \Gamma \vdash t : T_1 \shortmid R}{\Gamma \vdash s(t) : T_2 \shortmid P \lor Q \lor R} \qquad \dfrac{\begin{array}{c}\Gamma \mid Q \vdash s : \{P\}\ T_1 \to T_2 \qquad \Gamma \mid R \vdash t : T_1 \\ \Gamma \vdash P <: P \lor Q \lor R \qquad \Gamma \vdash Q <: P \lor Q \lor R \\ \Gamma \vdash R <: P \lor Q \lor R\end{array}}{\Gamma \mid P \lor Q \lor R \vdash s(t) : T_2}$$

Term applications just lift directly from λ_{FLX} to System $\mathsf{F}_{<:\mathsf{BE}}$ via (A-SUB-EFF) and (A-SUB).

$$\Gamma \vdash \mathtt{do}\ b : \mathtt{int} \shortmid b \qquad\qquad \Gamma \mid b \vdash \mathtt{do}\ b : \{\bot\}\ \mathtt{int}$$

$$\dfrac{\Gamma \vdash t : T \shortmid P \qquad P \land b \equiv_{\mathbb{B}} \bot}{\Gamma \vdash t\ \mathtt{without}\ b : T \shortmid P} \qquad\qquad \dfrac{\Gamma \mid P \vdash t : T \qquad \Gamma \vdash P <: \ {\sim} b}{\Gamma \mid P \vdash \mathtt{without}\ b\ t : T}$$

Effectful operations translate almost directly as well from λ_{FLX} to System $\mathsf{F}_{<:\mathsf{BE}}$; note that in any Boolean algebra, $A \sqsubseteq \neg B$ if and only if $A \sqcap B \sqsubseteq \bot$.

$$\dfrac{\begin{array}{c}\Gamma \vdash t : T_1 \shortmid P \\ P \equiv_{\mathbb{B}} Q \qquad T_1 \equiv_{\mathbb{B}} T_2\end{array}}{\Gamma \vdash t : T_2 \shortmid Q} \qquad\qquad \dfrac{\begin{array}{c}\Gamma \mid P \vdash t : T_1 \\ \Gamma \vdash P <: Q \qquad \Gamma \vdash T_1 <: T_2\end{array}}{\Gamma \mid Q \vdash t : T_2}$$

Type and effect equality in λ_{FLX} corresponds to a combination of (A-SUB-EFF) and (A-SUB).

$$\dfrac{\begin{array}{c}\Gamma \vdash s : \mathtt{int} \shortmid P \\ \Gamma \vdash t : T \shortmid Q \qquad \Gamma \vdash u : T \shortmid R\end{array}}{\Gamma \vdash \mathtt{if0}\ s\ \mathtt{then}\ t\ \mathtt{else}\ u : T \shortmid P \lor Q \lor R} \qquad \dfrac{\begin{array}{c}\Gamma \mid P \vdash s : \{P_{any}\}\ \mathtt{int} \\ \Gamma \mid Q \vdash t_1 : T \qquad \Gamma \mid R \vdash t_2 : T \\ \Gamma \vdash P <: P \lor Q \lor R \qquad \Gamma \vdash Q <: P \lor Q \lor R \\ \Gamma \vdash R <: P \lor Q \lor R\end{array}}{\Gamma \mid P \lor Q \lor R \vdash \mathtt{if0}\ s\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : T}$$

Conditionals in λ_{FLX} translate directly, except using (A-SUB-EFF) to limit the effects.

4.3 Implementation of λ_{FLX}

We have implemented abstraction-site subeffecting in the Flix compiler. Flix uses a constraint-based type and effect inference algorithm with equality constraints. Supporting subeffect constraints would require an intricate rewrite of the inference algorithm. Fortunately, and by design, we can rewrite a subeffect constraint like $P <: Q$ into an equality constraint $P \lor Y = Q$, where Y is a fresh effect variable. We can apply this rewrite at every lambda abstraction, which allows us to support abstraction-site subeffecting without extensive compiler modification.

Unfortunately, there is no free lunch. While adding an extra effect variable is simple, it could affect the complexity of the Boolean unifiers we must compute. We study this in Section 5 and find that the performance impact is acceptable. A trivial and effective optimization we perform is to omit subeffecting when a function is know to be pure. This is mostly relevant for top-level functions since they are required to be annotated in Flix.

4.4 Type and Effect Inference

A major advantage of the Flix type and effect system is that, because the effect language forms a Boolean algebra, we can support inference via *Boolean unification* [6, 30, 31]. In particular, Boolean unification is *unitary* [8]. That is, given two Boolean formulas T_1 and T_2, either there exists a most-general unifier S such that $S(T_1) \equiv_{\mathbb{B}} S(T_2)$, or T_1 and T_2 cannot be unified. While there may be many unifiers for T_1 and T_2, there is always a most-general unifier. The existence of computable most-general unifiers enables the implementation of unification-based type and effect inference.

We can compute Boolean unifiers using the *Successive Variable Elimination* (SVE) algorithm [6] or Löwenheim's method [31]. Boolean unification tends to produce exponentially large unifiers, so special care is needed to ensure acceptable performance. In any case, the key takeaway is that, for the Flix type and effect system in particular, and for any qualified system based on Boolean algebras in general, Boolean unification enables inference. That is part of what makes a qualifier system based on Boolean algebras attractive.

4.5 Abstraction-Site Subeffecting: Three Variants

The λ_{FLX} calculus supports subeffecting at every abstraction-site, i.e., at every lambda expression. The real Flix language, however, has three different kinds of functions:

- (SE-Def): Flix has *module-level function definitions*. For example, `Option.map` and `List.flatMap` are module-level function definitions. These functions have mandatory type and effect signatures. Applying subeffecting here means that the *inferred* effect of any function's body can be smaller than the *declared* effect of that function.
- (SE-Inst): Flix has type classes (called *traits*) and type class implementations (called *instances*). A trait defines one or more function signatures, which must be specified by each instance. For example, Flix has the `Foldable` trait, which specifies function signatures like `Foldable.foldLeft` and `Foldable.foldRight`, and is implemented by data structures such as `List[t]` and `MutList[t, r]`. Applying subeffecting here means that the effect of a function, in an instance, can be smaller than the declared effect from the function signature in the trait.
- (SE-Lam): Flix has regular lambda expressions that occur inside functions, e.g., inside module-level functions or inside trait instance function implementations. Applying subeffecting here is like in the calculus.

We have implemented abstraction-site subeffecting for SE-Def, SE-Ins, and SE-Lam, and we are now ready to evaluate it.

5 Evaluation

We evaluate the usefulness of extending Flix with abstraction-site subeffecting. Specifically, we consider the following four research questions:

- **RQ1**: How common are effect upcasts?
- **RQ2**: Which existing programming patterns can be improved by subeffecting?
- **RQ3**: How many effect upcasts can be elided with the abstraction-site subeffecting?
- **RQ4**: What is the performance impact of abstraction-site subeffecting on type inference?

We answer RQ3 and RQ4 for each variant of abstraction-site subeffecting: SE-Def, SE-Ins, and SE-Lam. We will study these questions on a *single* large benchmark: The Flix Standard Library.

5.1 The Benchmark: The Flix Standard Library

The Flix Standard Library[2] is the subject of the empirical evaluation. As Table 1 illustrates, it is one of the largest real-world libraries subject to a type and effect system[3]:

Table 1. Languages with effect systems and their standard libraries. Adapted from Madsen et al. [37].

	GitHub		*Lines of Code*		
Language	**Stars**	**Contributors**	**Library**	**⋆.lang**	**Repository**
Eff [3]	.8k	14	—	9 k	matijapretnar/eff
Frank [28]	.3k	6	.1 k	5 k	frank-lang/frank
Effekt [9, 10]	.3k	15	.5 k	14 k	effekt-lang/effekt
Links [21, 27]	.3k	30	1.4 k	31 k	links-lang/links
Koka [26, 52]	3.1k	30	17 k	110 k	koka-lang/koka
Flix [30, 35]	2.1k	68	39 k	149 k	flix/flix

The Library column shows the number of lines of code in the library for that language[4]. The ⋆.lang column shows the total number of lines of code in that language within the same repository.

Library Overview. Table 2 shows an overview of the modules in the library of at least 125 lines. The Defs column shows the total number of functions in the module. This includes top-level functions, public as well as private, but not lambda expressions. It also includes default trait functions and functions in trait instances. The Pure, Effectful, and Poly columns show the number of pure functions, functions with a concrete effect set, and effect polymorphic functions in the module, respectively. If a function has an effect like `ef + IO`, then it is counted as effect polymorphic.

5.2 RQ1: How Common are Effect Upcasts?

Flix lacks subeffecting; hence today programmers must insert explicit *effect upcasts* to make their programs pass the typechecker. Specifically, we identify two forms of upcasts:

First, there is *direct subeffecting*, where the programmer has a term t with effect P, i.e., $\Gamma \vdash t : T \mid P$, but wants it typed as $\Gamma \vdash t : T \mid Q$, where $P <: Q$. Such an effect upcast can be written as `checked_ecast(t)`, which instructs the compiler to add a free effect variable onto the actual effect of t. A `checked_ecast(t)` is always sound.

Second, there is *structural subeffecting*, where the programmer has a term t typed as:

$$\Gamma \vdash t : \mathtt{List[int} \xrightarrow{P} \mathtt{int]} \mid R$$

but wants it typed with Q instead of P, where $P <: Q$. In this case, subeffecting is required on the function type *inside* the list. Here, the programmer has to write: `unchecked_cast(t as ... )`, where the dots should be filled with the desired type. As the name suggests, the cast is not checked by the compiler, and hence may be unsound. As is turns out, the Flix Standard Library does not have any uses of structural subeffecting.

We count all uses of `checked_ecast` in the Flix Standard Library. Table 2 shows the results per module. In total, we find 47 occurrences of `checked_ecast`, i.e., the library has 47 uses of subeffecting. It is these explicit upcasts that we hope to get rid of with abstraction-site subeffecting.

[2]https://api.flix.dev/ and https://github.com/flix/flix/tree/master/main/src/library
[3]If Java's checked exceptions count as an effect system, then the Java Class Library is obviously the largest.
[4]The line count was measured with `find -name '*.ext'|xargs cat|wc -l` in the root directory and then in the directory which we identified as the standard library using the appropriate language extension.

Table 2. The Flix Standard Library.

			Effect Classification				*Effect Variables*			
Module	**Lines**	**Defs**	**Pure**	**Effectful**	**Poly**	**Upcasts**	**Baseline**	**SE-Def**	**SE-Ins**	**SE-Lam**
Adaptor.flix	238	21	6	5	10	0	137	+15	+0	+22
Applicative.flix	185	14	8	0	6	0	42	+6	+0	+20
Array.flix	1,689	138	5	0	133	1	1,702	+130	+3	+92
BigDecimal.flix	213	22	22	0	0	0	49	+0	+0	+0
BigInt.flix	393	37	37	0	0	0	117	+0	+0	+10
Chain.flix	975	112	59	2	51	1	658	+36	+17	+54
Chalk.flix	560	55	55	0	0	0	295	+0	+0	+7
Char.flix	243	31	31	0	0	0	38	+0	+0	+0
CodePoint.flix	306	33	33	0	0	0	72	+0	+0	+0
Concurrent/…	727	31	3	28	0	0	151	+28	+0	+6
DelayList.flix	1,346	110	55	1	54	3	943	+42	+13	+78
DelayMap.flix	889	81	37	1	43	0	588	+38	+6	+177
Environment.flix	137	16	16	0	0	0	31	+0	+0	+0
Eq.flix	242	37	37	0	0	0	129	+0	+0	+0
File.flix	452	37	1	29	7	0	113	+35	+1	+34
Files.flix	943	47	1	35	11	4	496	+46	+0	+56
Fixpoint/…	3,243	152	123	0	29	0	2,401	+29	+0	+225
Float32.flix	339	37	37	0	0	0	118	+0	+0	+0
Float64.flix	373	39	39	0	0	0	117	+0	+0	+0
Foldable.flix	396	46	0	0	46	2	264	+46	+0	+74
GetOpt.flix	421	26	24	0	2	0	188	+2	+0	+19
Graph.flix	558	29	5	0	24	0	261	+24	+0	+13
Group.flix	165	29	29	0	0	0	74	+0	+0	+0
Hash.flix	171	23	23	0	0	0	112	+0	+0	+0
Identity.flix	133	20	10	0	10	1	55	+0	+10	+1
Int16.flix	446	54	54	0	0	0	172	+0	+0	+0
Int32.flix	480	56	56	0	0	0	171	+0	+0	+1
Int64.flix	494	57	57	0	0	0	188	+0	+0	+1
Int8.flix	419	52	52	0	0	0	159	+0	+0	+0
Iterator.flix	759	50	12	0	38	4	692	+38	+0	+73
JoinLattice.flix	212	14	14	0	0	0	59	+0	+0	+0
List.flix	1,433	155	89	2	64	6	995	+49	+17	+128
Map.flix	984	121	50	2	69	0	655	+58	+13	+174
MeetLattice.flix	213	14	14	0	0	0	59	+0	+0	+0
MultiMap.flix	568	71	31	1	39	0	412	+33	+7	+122
MutDeque.flix	446	41	5	0	36	0	557	+35	+1	+18
MutDisjointSets.flix	178	10	1	0	9	0	112	+9	+0	+5
MutList.flix	936	82	1	0	81	0	1,068	+78	+3	+51
MutMap.flix	679	78	0	0	78	3	555	+75	+3	+22
MutQueue.flix	225	18	0	0	18	0	257	+17	+1	+8
MutSet.flix	436	50	0	0	50	1	362	+49	+1	+7
Nec.flix	1,034	129	63	2	64	8	919	+42	+24	+95
Nel.flix	712	110	54	2	54	1	426	+32	+24	+25
Option.flix	585	78	33	0	45	2	242	+28	+17	+16
Order.flix	653	71	71	0	0	0	263	+0	+0	+0
PartialOrder.flix	197	14	14	0	0	0	56	+0	+0	+0
Prelude.flix	247	18	12	2	4	0	59	+6	+0	+3
RedBlackTree.flix	1,003	80	41	0	39	3	722	+28	+11	+95
Reducible.flix	296	38	21	0	17	2	197	+17	+0	+64
Regex.flix	716	55	42	0	13	0	457	+13	+0	+16
Result.flix	443	48	17	0	31	2	183	+28	+3	+16
SemiGroup.flix	169	25	25	0	0	0	68	+0	+0	+0
Set.flix	642	83	43	1	39	0	465	+30	+10	+103
String.flix	1,377	121	100	0	21	0	783	+21	+0	+53
StringBuilder.flix	147	17	0	0	17	0	101	+17	+0	+6
Time/…	176	15	10	5	0	0	21	+5	+0	+0
ToString.flix	186	27	27	0	0	0	129	+0	+0	+0
Validation.flix	294	32	14	0	18	0	140	+16	+2	+20
Vector.flix	1,523	158	87	3	68	3	1,162	+53	+18	+104
…	…	…	…	…	…	…	…	…	…	…
Totals	37,551	3,543	2,027	133	1,383	47	22,871	+1,311	+205	+2,142

5.3 RQ2: Programming Patterns That Rely on Subeffecting

We now investigate the programming patterns that give rise to effect upcasts. The following examples are presented without subeffecting.

Example I. We have already seen the effect upcast in `List.foldRight` in Section 2:

```
def foldRight(f: (a, b) -> b \ ef, s: b, l: List[a]): b \ ef =
    def loop(ll, k) = match ll {
        case Nil     => k(s)
        case x :: xs => loop(xs, ks -> k(f(x, ks)))
    };
    loop(l, x -> checked_ecast(x))
```

The `foldRight` function is effect polymorphic; its effect depends on the effect of its function argument `f`. Internally, the `foldRight` function is implemented using a recursive function with an accumulator, a typical pattern in functional programming to ensure that the stack does not overflow. The accumulator is a function that is built up and then called in the base case. Notably, the accumulator function calls `f`, which means it has the effect `ef`. More precisely, the type of `k` is `b -> b \ ef`. The computation starts by calling `loop` with the identity function. *However*, this does not work because the identity function is pure, which is incompatible with the requirement that the continuation must have the effect `ef`. An effect upcast (`checked_ecast`) casts the pure variable expression `x` to have the effect `ef`. Unfortunately, if we forgot this effect upcast, the Flix compiler would tell us:

```
>> Mismatched pure and effectful Functions.
4 |         case x :: xs => loop(xs, ks -> k(f(x, ks)))
                                          ^^^^^^^^^^^
                                          mismatched effects.
```

Notably, the error occurs at the “wrong” source location, i.e., at the call to the continuation, and *not* where we have to insert the effect upcast! Anecdotally, programmers are very confused by such errors, and often cannot progress without help.

Example II. We find another effect upcast in the `Iterable` instance for `Array[a, r]`:

```
instance Iterable[Array[a, r]] {
    type Elm = a // Associated type.
    type Aef = r // Associated effect.
    def iterator(rc: Region[r1], a: Array[a, r]):
                                Iterator[a, r + r1, r1] \ (r + r1) =
        checked_ecast(Array.iterator(rc, a))
}
```

Here, `iterator` returns a fresh iterator over the elements of the array. The effect upcast is required because the declared signature of the `iterator` function in the `Iterable` trait *must* have effect `r + r1`. That is, when we create an iterator, we expect a heap effect in the region of the iterator (`r1`) *and* a heap effect in the region of the mutable memory (`r`). However, uniquely in the case of `Array[a, r]`, creating the iterator does not need to touch region `r`. The reason is that to create the iterator, we need the length of the array, but accessing the length of a mutable array is pure, because arrays cannot change their length once created. Hence we must upcast the effect from `r1` to `r + r1`.

Example III. We find another effect upcast in the `Foldable.foldLeftM` function:

```
def foldLeftM(f: (b, a) -> m[b] \ ef, s: b, t: t[a]):
                                m[b] \ (ef + Foldable.Aef[t]) with Monad[m] =
    let f1 = (x, acc) -> z -> Monad.flatMap(acc, f(z, x));
    s |> Foldable.foldRight(f1, x1 -> checked_ecast(Applicative.point(x1)), t)
```

The `foldLeftM` function performs an effectful monadic fold over `t`, which is `Foldable`. The function argument `f` has effect `ef`. Moreover, the data structure `t` has an *associated effect* `Foldable.Aef[t]` (see [29]). The problem here is that the function `f1` has effect `ef`, but the initial function passed to `foldRight`, i.e., `x1 -> Applicative.point(x1)`, is pure. Hence the need for the effect upcast.

This function serves to illustrate the richness and non-trivial complexity of the Flix type and effect system and the Flix Standard Library. Specifically, `foldLeftM` uses all of the following programming language features: (i) type classes, (ii) higher-kinded types, (iii) effect polymorphism, (iv) associated effects, *and now* (v) abstraction-site subeffecting, which can be used to omit the effect upcast.

Two of these examples require effect upcasts because of a continuation-based programming style. This programming pattern is pervasive and is the cause of the vast majority of effect upcasts in the library.

5.4 RQ3: Effect Upcasts That Can Be Elided With Subeffecting

We now measure how many effect upcasts are removable by each variant of abstraction-site subeffecting: SE-Def, SE-Ins, and SE-Lam. Table 3 shows the result.

Table 3. Breakdown of removable effect upcasts in the Flix Standard Library.

	Removable Upcasts		
Upcasts	**SE-Def**	**SE-Ins**	**SE-Lam**
47	0	1	46

Table 3 shows that of the 47 upcasts, SE-Def eliminates 0 casts, SE-Ins eliminates 1 cast, and SE-Lam eliminates 46. Hence, together, SE-Ins and SE-Lam eliminate *all effect upcasts*. *All* uses of `checked_ecast` can be removed.

5.5 RQ4: Performance Impact of Subeffecting

Abstraction-site subeffecting is based on introducing new effect "slack" variables, which allow the effect of a function to be widened. Recall that the three variants: SE-Def, SE-Ins, and SE-Lam differ in where these variables are introduced. Unfortunately, introducing new effect variables is not free: it increases the complexity of the unifiers computed by Boolean unification during type and effect inference, and hence has an impact on performance.

We conduct a preliminary experiment to measure the cost of SE-Def, SE-Ins, and SE-Lam. We measure the increase in effect variables and the direct impact on compiler throughput. The experiments use Flix version 0.52.0 extended with the three variants. Flix is run using Java 21.0.2 on an Intel i5-13500 CPU with 64 GB of memory. We compute the throughput by running the compiler on the Flix Standard Library and its unit tests and report the median of 250 runs.

Table 2 shows a detailed breakdown of the increase in effect variables per module and per variant. For example, the row for the `List` module shows that without subeffecting, type and effect inference introduces 995 effect variables. Adding subeffecting, in the form of SE-Def, SE-Ins, and SE-Lam, introduces 49, 17, and 128 additional effect variables, respectively.

Table 4 shows a summary of the performance results. We see that each of SE-Def, SE-Ins, and SE-Lam incurs a modest slowdown due to the increased complexity of Boolean unification. We remark that these results are preliminary and that optimization possibilities are left on the table.

Table 4. Performance cost of abstraction-site subeffecting.

	Effect Variables		*Performance*	
Variants	**Variables**	**Increase**	**Throughput**	**Slowdown**
Baseline	22,871	-	101,473 lines/sec	-
SE-Def	24,182	+5.7%	96,796 lines/sec	1.05x
SE-Ins	23,076	+0.9%	99,975 lines/sec	1.01x
SE-Lam	25,013	+9.4%	97,503 lines/sec	1.04x

The point of our experiment is to demonstrate that inference with abstraction-site subeffecting is practical despite the increase in variables.

5.6 Lessons Learned

We conclude the evaluation with the following observations:

- **RQ1**: We find that – due to the lack of subeffecting in Flix – effect upcasts are common throughout the Flix Standard Library.
- **RQ2**: We find that the programming patterns that rely on effect upcasts typically involve recursive functions that build an explicit effectful continuation to avoid blowing the stack.
- **RQ3**: We count that, of the 47 effect upcasts, abstraction-site subeffecting eliminates 0 casts with SE-Def, 1 cast with SE-Ins, and 46 casts with SE-Lam. This strongly suggests that subeffecting is important for lambda expressions, but less important for module-level definitions and instance definitions. Moreover, the fact that SE-Ins and SE-Lam *eliminate all effect upcasts* suggests that general subeffecting is not needed since we would assume that the standard library has more effect polymorphism than the average program.
- **RQ4**: Performance experiments suggest that SE-Def, SE-Ins, and SE-Lam incur slowdowns, but that the performance of type and effect inference remains acceptable.

6 Related Work

Lattice-Based Type Qualifiers. Our work on System $\mathsf{F}_{<:\mathsf{B}}$ is a natural extension of the free lattice structure that Lee et al. [25] present in System $\mathsf{F}_{<:\mathsf{Q}}$. Boolean algebras are after all just lattices equipped with negation and distributivity axioms. System $\mathsf{F}_{<:\mathsf{Q}}$ in turn ties together common subqualification structures seen in previous work, such as Boruch-Gruszecki et al. [7]'s capturing types for Scala, Wei et al. [51]'s polymorphic reachability types, and Gariano et al. [17]'s polymorphic effect system. While most previous work in this area falls neatly in the structure of a (free) lattice first presented by Foster et al. [15], some does not. Notably, Wei et al. [51] introduce the notion of freshness in their reachability calculus to separate freshly allocated values, which by definition cannot be reachable by existing values in a program's state. We think that their notion of freshness can be modelled by negation in a Boolean algebra, and it would be interesting future work to model reachability using these ideas. Other work on type qualifiers includes a long line of work on other invariants such as immutability [22, 49, 53], and other invariants [12]. It would be interesting to see how negation can be used to augment these qualifier systems. Similar work has also been done recently to model general polymorphic effect systems using algebraic structures [19].

Boolean Algebras Without Subqualification. Dually to work on subtyping with type qualifiers, Boolean algebras and formulas have also been used to enrich types to capture additional invariants about a program. Unlike the work on type qualifiers, which has a long history, the work done on Boolean algebras has been more recent. For example, only recently have Boolean algebras been used to model (non)-nullable references [36] and effects [35], in comparison to a long line of work done on nullability in the context of type qualifiers. As Madsen and van de Pol note, part of the limitations of their work is that subqualification on Boolean qualifiers was an open problem [36, Conjecture 4.18]. We answer their conjecture affirmatively, and hope that our work makes Boolean algebra based type qualifiers more accessible to other language designers going forward.

Boolean-Like Qualifier Systems. Outside of systems that use Boolean algebras, there are also systems that have notions of exclusion in their type systems. For example, Qi and Myers [44] propose a type and effect system for reasoning about when fields can be safely read in a mutable object oriented language. They use a notion of exclusion to qualify object types with what fields are available to be read, and an effect system on field writes to make fields accessible once written to. It would be interesting work to replicate their initialization system using ideas from this paper, restrictable variants and effect exclusion.

Boolean Unification Algorithms. Inference of Boolean effects requires Boolean unification algorithms for solving the subqualification constraints that are generated as part of the type checking process. Boolean formula unification algorithms date back as far as Boolean values themselves, as seen in Boole's [6] original work, and Löwenheim's [31] turn of the century book. More modern work includes that of Rudeanu [45] and Buttner and Simonis [11]. Boudet et al. [8] investigate unification on general Boolean algebras and rings. Finally, of particular interest to implementers will be Baader [2]'s study of the computational complexity of Boolean unification algorithms.

Algebraic Subtyping. Closely related to lattices and Boolean algebras is recent work on *algebraic subtyping*, first by Dolan and Mycroft [13] and more recently by Parreaux [42] and Parreaux and Chau [43]. In contrast to our approach, where we lightly augment the existing subtyping lattice of System $\mathsf{F}_{<:}$ with support for Boolean-algebra valued type qualifiers, the literature on algebraic subtyping is focussed on integrating the algebraic theory of lattices into the subtyping structure directly. While their approach is more expressive – for example, with records, they are able to reason about intersections, unions, and negations of records – we only propose using qualifiers to indicate which fields are present. Unions and intersections on types add expressiveness but also add complexity. The changes needed to support a full Boolean algebra based subtyping lattice are more heavyweight than what we propose here. Concretely, they have algebraic rules that go beyond Boolean algebras, which require a custom solver.

Effect Systems and Type Qualifiers. Effect systems augment type systems to track the side effects that a given computation can perform. They describe properties of *computation*, dually to how type qualifiers describe properties of *values*. In the presence of functions as first-class values, this distinction can be blurry; some effect systems can be implemented using type qualifiers as a building block. This is what we did with System $\mathsf{F}_{<:\mathsf{BE}}$, where we used annotated type qualifiers on function types to describe the effects that a function can perform when called. However, while closely related – type qualifiers can be used to implement some effect systems, and effects can be described by lattices [46] – there are effect systems that use different implementation techniques. Some, like Leijen [26], use *row polymorphism* [50] to handle effect polymorphism and limited subeffecting. Others, as we describe below, use different algebraic structures to describe *noncommutative* effects. Moreover, an effect system needs additional introduction and elimination forms to relate the qualifiers on function types to the actual effects that are performed in the body of a function.

Ordered Effects. Effect systems can come with other algebraic structures as well. In many situations, one wants to keep track of the order in which effects happen to be performed, in addition to which effects were performed. Examples of such tasks involve atomicity or synchronization, as locks can only be unlocked *after* being locked. For this task, Gordon [19] proposed a system which used *monoids* or *semigroups* instead of the full commutative structure of a lattice or Boolean algebra.

Abstraction-Site Subeffecting. Abstraction-site subeffecting has been used in practice by other systems to handle type inference around sub-effecting, as we do in our contribution to Flix. Systems which use abstraction-site subeffecting (even though they do not use the term abstraction-site subeffecting) include Bierman and Parkinson [4], Bocchino et al. [5], Gordon [18], Greenhouse and Boyland [20], Talpin and Jouvelot [47].

7 Conclusion

We have presented a calculus System $\mathsf{F}_{<:\mathsf{B}}$ that extends System $\mathsf{F}_{<:}$ with type qualifiers over Boolean algebras and with support for qualifier polymorphism and subqualification. We have shown how System $\mathsf{F}_{<:\mathsf{B}}$ can be used as a design recipe for a type and effect system, System $\mathsf{F}_{<:\mathsf{BE}}$, with effect polymorphism, subeffecting, and polymorphic effect exclusion. We have mechanized the calculi in Rocq. Building on System $\mathsf{F}_{<:\mathsf{B}}$, we have introduced λ_{FLX} to establish formal foundations of the Flix type and effect system. We have described *abstraction-site subeffecting* and discussed three variants: SE-Def, SE-Ins, and SE-Lam, which differ in where subeffecting is applied. We have implemented all three variants in the Flix compiler. Using SE-Ins and SE-Lam, we are able to eliminate all effect upcasts in the Flix Standard Library. The upshot is that Flix now has a type and effect system, established on formal foundations, which supports effect polymorphism, subeffecting, and polymorphic effect exclusion.

Acknowledgments

We thank the anonymous OOPSLA reviewers for their helpful feedback. We thank Ross Willard for his useful insights into free lattices and boolean algebras. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada, the Danmarks Frie Forskningsfond, and an Ontario Graduate Scholarship.

Data-Availability Statement

The artifact for this paper is available on both Software Heritage [23] and on Zenodo [24]. It includes both the Rocq mechanization of System $\mathsf{F}_{<:\mathsf{B}}$ and System $\mathsf{F}_{<:\mathsf{BE}}$ and our modified Flix compiler with support for abstraction-site subeffecting, with instructions for using both.

References

[1] Brian Aydemir, Arthur Charguéraud, Benjamin C. Pierce, Randy Pollack, and Stephanie Weirich. 2008. Engineering formal metatheory. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. doi:10.1145/1328438.1328443 14

[2] Franz Baader. 1998. On the complexity of Boolean unification. *Inform. Process. Lett.* 67, 4 (1998). doi:10.1016/s0020-0190(98)00106-9 24

[3] Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015). doi:10.1016/j.jlamp.2014.02.001 19

[4] G. M. Bierman and M. J. Parkinson. 2003. Effects and effect inference for a core Java calculus. *Electronic Notes in Theoretical Computer Science* 82, 8 (2003). doi:10.1016/s1571-0661(04)80803-x 25

[5] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. doi:10.1145/1640089.1640097 25

[6] George Boole. 1847. *The mathematical analysis of logic*. 18, 24

[7] Aleksander Boruch-Gruszecki, Martin Odersky, Edward Lee, Ondřej Lhoták, and Jonathan Brachthäuser. 2023. Capturing Types. *ACM Transactions on Programming Languages and Systems* 45, 4 (2023). doi:10.1145/3618003 1, 23

[8] Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauss. 1989. Unification in Boolean Rings and Abelian Groups. *Journal of Symbolic Computation* 8, 5 (1989). doi:10.1016/s0747-7171(89)80054-9 18, 24

[9] Jonathan Immanuel Brachthäuser, Philipp Schuster, Edward Lee, and Aleksander Boruch-Gruszecki. 2022. Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back. *Proceedings of the ACM on Programming Languages* 6, OOPSLA1 (2022). doi:10.1145/3527320 1, 13, 19

[10] Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effects as capabilities: effect handlers and lightweight effect polymorphism. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428194 19

[11] Wolfram Buttner and Helmut Simonis. 1987. Embedding boolean expressions into logic programming. *Journal of Symbolic Computation* 4, 2 (1987). doi:10.1016/s0747-7171(87)80065-2 24

[12] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*. doi:10.1145/1985793.1985889 23

[13] Stephen Dolan and Alan Mycroft. 2017. Polymorphism, subtyping, and type inference in MLsub. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. doi:10.1145/3009837.3009882 24

[14] M. Felleisen and D. P. Friedman. 1987. A calculus for assignments in higher-order languages. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*. doi:10.1145/41625.41654 11, 13

[15] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. 1999. A theory of type qualifiers. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. doi:10.1145/301618.301665 1, 7, 8, 10, 23

[16] Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*. 5

[17] Isaac Oscar Gariano, James Noble, and Marco Servetto. 2019. Callε: an effect system for method calls. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. doi:10.1145/3359591.3359731 23

[18] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. doi:10.4230/LIPICS.ECOOP.2017.13 25

[19] Colin S. Gordon. 2021. Polymorphic Iterable Sequential Effect Systems. *ACM Transactions on Programming Languages and Systems* 43, 1 (2021). doi:10.1145/3450272 23, 25

[20] Aaron Greenhouse and John Boyland. 1999. *An Object-Oriented Effects System*. Springer Berlin Heidelberg. doi:10.1007/3-540-48743-3_10 25

[21] Daniel Hillerström and Sam Lindley. 2016. Liberating effects with rows and handlers. In *Proceedings of the 1st International Workshop on Type-Driven Development*. doi:10.1145/2976022.2976033 19

[22] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. 2012. Reim & ReImInfer: checking and inference of reference immutability and method purity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. doi:10.1145/2384616.2384680 23

[23] Edward Lee, Ondřej Lhoták, Jonathan Lindegaard Starup, and Magnus Madsen. 2025. *Qualified Types with Boolean Algebras (Artifact)*. 2, 25

[24] Edward Lee, Ondřej Lhoták, Jonathan Lindegaard Starup, and Magnus Madsen. 2025. *Qualified Types with Boolean Algebras (Artifact)*. doi:10.5281/zenodo.16915676 2, 25

[25] Edward Lee, Yaoyu Zhao, Ondřej Lhoták, James You, Kavin Satheeskumar, and Jonathan Immanuel Brachthäuser. 2024. Qualifying System $F_{<:}$: Some Terms and Conditions May Apply. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 (2024). doi:10.1145/3649832 1, 2, 3, 7, 8, 9, 10, 11, 12, 13, 14, 16, 23

[26] Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. *Electronic Proceedings in Theoretical Computer Science* 153 (2014). doi:10.4204/eptcs.153.8 19, 24

[27] Sam Lindley and James Cheney. 2012. Row-based effect types for database integration. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation*. doi:10.1145/2103786.2103798 19

[28] Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do be do be do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. doi:10.1145/3009837.3009897 19

[29] Matthew Lutze and Magnus Madsen. 2024. Associated Effects: Flexible Abstractions for Effectful Programming. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024). doi:10.1145/3656393 14, 22

[30] Matthew Lutze, Magnus Madsen, Philipp Schuster, and Jonathan Immanuel Brachthäuser. 2023. With or Without You: Programming with Effect Exclusion. *Proceedings of the ACM on Programming Languages* 7, ICFP (2023). doi:10.1145/3607846 1, 2, 3, 14, 15, 16, 18, 19

[31] Leopold Löwenheim. 1908. *Über das Auflösungsproblem im logischen Klassenkalkul*. 18, 24

[32] Cong Ma, Zhaoyi Ge, Edward Lee, and Yizhou Zhang. 2024. Lexical Effect Handlers, Directly. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024). doi:10.1145/3689770 1

[33] Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the masses: programming with first-class Datalog constraints. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428193 14
[34] Magnus Madsen, Jonathan Lindegaard Starup, and Matthew Lutze. 2023. Restrictable Variants: A Simple and Practical Alternative to Extensible Variants. doi:10.4230/LIPICS.ECOOP.2023.17 2, 3, 5, 6
[35] Magnus Madsen and Jaco van de Pol. 2020. Polymorphic types and effects with Boolean unification. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020). doi:10.1145/3428222 1, 2, 3, 14, 19, 24
[36] Magnus Madsen and Jaco van de Pol. 2021. Relational nullable types with Boolean unification. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021). doi:10.1145/3485487 24
[37] Magnus Madsen, Jaco van de Pol, and Troels Henriksen. 2023. Fast and Efficient Boolean Unification for Hindley-Milner-Style Type and Effect Systems. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622816 19
[38] Daniel Marino and Todd Millstein. 2009. A generic type-and-effect system. In *Proceedings of the 4th international workshop on Types in language design and implementation.* doi:10.1145/1481861.1481868 1
[39] Shane Markstrum, Daniel Marino, Matthew Esquivel, Todd Millstein, Chris Andreae, and James Noble. 2010. JavaCOP: Declarative pluggable types for Java. *ACM Transactions on Programming Languages and Systems* 32, 2 (2010). doi:10.1145/1667048.1667049 1
[40] Marius Müller, Philipp Schuster, Jonathan Lindegaard Starup, Klaus Ostermann, and Jonathan Immanuel Brachthäuser. 2023. From Capabilities to Regions: Enabling Efficient Compilation of Lexical Effect Handlers. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2 (2023). doi:10.1145/3622831 1
[41] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jeff H. Perkins, and Michael D. Ernst. 2008. Practical pluggable types for Java. In *Proceedings of the 2008 international symposium on Software testing and analysis.* doi:10.1145/1390630.1390656 1
[42] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). doi:10.1145/3409006 24
[43] Lionel Parreaux and Chun Yin Chau. 2022. MLstruct: principal type inference in a Boolean algebra of structural types. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022). doi:10.1145/3563304 24
[44] Xin Qi and Andrew C. Myers. 2009. Masked types for sound object initialization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* doi:10.1145/1480881.1480890 24
[45] Sergiu Rudeanu. 1974. *Boolean functions and equations.* 24
[46] Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. *Lightweight Polymorphic Effects.* Springer Berlin Heidelberg. doi:10.1007/978-3-642-31057-7_13 24
[47] Jean-Pierre Talpin and Pierre Jouvelot. 1992. Polymorphic type, region and effect inference. *Journal of Functional Programming* 2, 3 (1992). doi:10.1017/s0956796800000393 15, 25
[48] Cassia Torczon, Emmanuel Suárez Acevedo, Shubh Agrawal, Joey Velez-Ginorio, and Stephanie Weirich. 2024. Effects and Coeffects in Call-by-Push-Value. *Proceedings of the ACM on Programming Languages* 8, OOPSLA2 (2024). doi:10.1145/3689750 1
[49] Matthew S. Tschantz and Michael D. Ernst. 2005. Javari: adding reference immutability to Java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* doi:10.1145/1094811.1094828 23
[50] Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS'87), Ithaca, New York, USA, June 22-25, 1987.* 24
[51] Guannan Wei, Oliver Bračevac, Songlin Jia, Yuyan Bao, and Tiark Rompf. 2024. Polymorphic Reachability Types: Tracking Freshness, Aliasing, and Separation in Higher-Order Generic Programs. *Proceedings of the ACM on Programming Languages* 8, POPL (2024). doi:10.1145/3632856 1, 23
[52] Ningning Xie, Jonathan Immanuel Brachthäuser, Daniel Hillerström, Philipp Schuster, and Daan Leijen. 2020. Effect handlers, evidently. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020). doi:10.1145/3408981 19
[53] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kiezun, and Michael D. Ernst. 2007. Object and reference immutability using Java generics. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* doi:10.1145/1287624.1287637 23

Received 2025-03-25; accepted 2025-08-12